Opérateurs arithmétiques matériels optimisés

Romain Michard

N° d'ordre : 465
N° attribué par la bibliothèque : 07ENSL0 465

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

spécialité : Informatique

Laboratoire de l'Informatique du Parallélisme

École doctorale de Mathématiques et d'Informatique fondamentale

présentée et soutenue publiquement le 25/06/2008 par

Monsieur Romain MICHARD

Titre :

Opérateurs arithmétiques matériels optimisés

Directeur de thèse :	Monsieur Arnaud TISSERAND	
Après avis de	Monsieur Neil BURGESS	*Membre/Rapporteur*
	Monsieur Daniel ETIEMBLE	*Membre/Rapporteur*
	Monsieur Olivier SENTIEYS	*Membre/Rapporteur*
Devant la commission d'examen formée de :		
	Monsieur Neil BURGESS	*Membre/Rapporteur*
	Monsieur Daniel ETIEMBLE	*Membre/Rapporteur*
	Monsieur Bernard GOOSSENS	*Membre*
	Monsieur Jean-Michel MULLER	*Membre*
	Monsieur Olivier SENTIEYS	*Membre/Rapporteur*
	Monsieur Arnaud TISSERAND	*Membre*

Un ordinateur fait au bas mot un million d'opérations à la seconde,
mais il a que ça à faire aussi.

Jean-Marie Gourio, *Brèves de comptoir*

Table des matières

Remerciements

Je voudrais remercier, avant tout, Arnaud Tisserand qui a été bien plus qu'un directeur de thèse. Les conditions n'ont pas été les meilleures mais il a toujours fait preuve de compréhension et de sympathie. Il m'a facilité les choses à chaque fois qu'il en a eu l'occasion et m'a constamment fait une confiance absolue qui a participé au climat dans lequel j'ai évolué. Ses conseils ont toujours été une mine précieuse.

Je remercie mes relecteurs. Merci au président du jury, et à ses autres membres qui sont venus de loin.

Je remercie le directeur de l'équipe Arénaire, Gilles Villard, et les deux directeurs successifs du LIP, Jean-Michel Muller et Frédéric Desprez, pour l'attention dont ils ont su faire preuve et les coups de main qu'ils ont pu me donner.

Corinne, Nathalie, Sylvie, Christoph, Claude-Pierre, Dominique, Florent, Guillaume, Nicolas, Serge, Sylvain et Vincent ont su créer une atmosphère particulièrement agréable, qu'ils en soient remerciés.

Je remercie chaleureusement Nicolas Veyrat–Charvillon pour ces années partagées, tant au niveau scientifique qu'au niveau personnel.

Enfin, j'adresse un remerciement particulier à Delphine pour m'avoir supporté (quelque soit l'acception de ce mot) et à Arthur pour le rayon de soleil qui m'éblouit à chacun de ses sourires.

Introduction

Contexte

L'informatique est une discipline plutôt récente (environ 70 ans selon [11]). C'est un domaine qui a connu une très vive évolution et qui la connaît encore aujourd'hui. Il semble assez difficile de parler de cette évolution sans parler de la loi de Moore [12]. En outre, cette loi se retrouve dans presque tous les documents traitant de l'informatique dans un sens large et la passer sous silence relève de la gageure. Cette loi constate expérimentalement, dès 1975, que le nombre de transistors intégrables dans les microprocesseurs double tous les deux ans environ. Depuis, elle se révèle étonnamment proche de la réalité même si cette progression ralentit légèrement ces dernières années et atteindra sans doute des limites technologiques. Néanmoins, ces limites ne cessent d'être reculées au fur et à mesure des améliorations scientifiques et des découvertes qui chamboulent ce qu'on croit constituer un mur infranchissable.

Il faut noter que le terme de « performance » en matière de circuit intégré numérique peut revêtir plusieurs aspects. On peut parler de fréquence d'utilisation ou de débit des informations (à maximiser) ou encore de consommation électrique (à minimiser). La surface du circuit n'est pas une performance à proprement parler mais elle a une implication directe sur le coût et elle reste donc un paramètre qu'on cherche à minimiser en permanence. Ces trois aspects concentrent un intérêt particulier et les chercheurs, entre autres, essayent d'améliorer ces performances.

Il est alors intéressant d'optimiser la conception de ces circuits, mais ils sont de plus en plus complexes et cette conception nécessite qu'on y consacre de plus en plus de temps. Il semble alors utile de s'appuyer sur des outils qui facilitent et accélèrent la conception. C'est ce type d'outils que nous avons cherché à fournir en créant des générateurs d'opérateurs arithmétiques optimisés, ce qui est un excellent moyen d'obtenir des descriptions efficaces d'opérateurs. Les raisons qui nous font préférer cette méthode à l'utilisation de bibliothèques sont exposées page 5.

Si on se penche sur les capacités de traitement d'un microprocesseur, ou d'autres circuits intégrés numériques, on voit essentiellement des fonctions logiques (inversion bit à bit d'un mot, par exemple) ou arithmétiques (addition de deux mots représentant deux nombres, etc.). L'arithmétique des ordinateurs est donc essentielle à optimiser. Il y a aussi des parties constituant la mémoire, le contrôle ou les entrées/sorties mais celles-ci sont moins concernées par nos travaux.

Ce document est composé d'une introduction assez complète donnant les bases à avoir en matière d'arithmétique des ordinateurs puis des 8 articles publiés dans le cadre de ces travaux entre 2004 et 2008.

Représentation des nombres

Les notions qui sont données ici sont détaillées et très bien expliquées dans deux ouvrages [13, 14] qui pourront être consultés pour davantage de précisions.

La valeur v d'un nombre entier ou réel approché est codée à l'aide d'un ensemble de n bits notés $\{v_{n-1}, \cdots, v_0\}$. Dans ce document, on s'intéresse à des nombres entiers ou virgule fixe (des nombres fractionnaires binaires). De plus, par souci de simplicité d'écriture et de clarté des explications, on ne traitera que la base 2. Néanmoins, le passage à d'autres bases est assez direct, en particulier pour les puissances de 2. Pour lire v, on établit une bijection entre les valeurs booléennes, codées électriquement par chaque fil, et l'ensemble de chiffres $\{0, 1\}$ pour chacun des v_i. Il existe différentes manières d'interpréter la valeur de ces fils pour en déduire la valeur mathématique v.

Les formats de représentation présentés ci-dessous permettent de coder des entiers. Si on veut travailler sur des approximations de réels, on introduit un facteur d'échelle $\text{ULP}(v)$ (pour *Unit in Last Position*) par lequel on multiplie l'entier codé pour obtenir v. Ce facteur est une puissance de 2 constante pour une notation donnée, qui indique le poids du bit le plus faible. Par exemple, $\text{ULP}(v)$ vaut 1 si on travaille sur des entiers. Si $\text{ULP}(v) = 2^{-2}$, alors on sait que les valeurs v ont deux bits après la virgule, c'est la partie fractionnaire. On est ainsi capable d'attribuer une valeur entière ou virgule fixe à un vecteur de valeurs logiques codées électriquement.

– *La notation non signée* ou *numération simple de position* représente une valeur positive v par :

$$v = \text{ULP}(v) \sum_{i=0}^{n-1} v_i \, 2^i,$$

$$\text{d'où } v \in \left([0, 2^n - 1] \cap \mathbb{N}\right) \times \text{ULP}(v).$$

– *La notation signe–valeur absolue* est une extension directe de la notation non signée aux nombres relatifs. On adjoint simplement en tête un bit représentant le signe du nombre à la représentation non signée de sa valeur absolue. Une valeur v est codée par :

$$v = (-1)^{v_{n-1}} \times \text{ULP}(v) \sum_{i=0}^{n-2} v_i \, 2^i, \text{ où } \begin{cases} v \geq 0 \text{ si } v_{n-1} = 0 \\ v \leq 0 \text{ si } v_{n-1} = 1 \end{cases},$$

$$\text{d'où } v \in \left([-2^{n-1} + 1, 2^{n-1} - 1] \cap \mathbb{N}\right) \times \text{ULP}(v).$$

Cette notation possède une double représentation de la valeur 0 ($+0$ et -0).

– *Le complément à 2* est une autre notation des nombres relatifs. Une valeur v est codée en complément à 2 par :

$$v = \text{ULP}(v) \left(-2^{n-1}\ v_{n-1} + \sum_{i=0}^{n-2} v_i\ 2^i \right),$$

$$\text{d'où } v \in \left([-2^{n-1}, 2^{n-1} - 1] \cap \mathbb{N} \right) \times \text{ULP}(v).$$

Bien que moins intuitif que la notation signe–valeur absolue, le complément à 2 résout un problème inhérent à l'écriture précédente. Pour pouvoir additionner deux nombres relatifs en signe–valeur absolue, il faudra un algorithme d'addition si les deux nombres sont de même signe, et un algorithme de soustraction sinon. Dans la représentation en complément à 2, l'algorithme d'addition reste le même (c'est en fait l'addition modulo 2^n) quels que soient les signes des opérandes. D'autre part, les additionneurs et soustracteurs y sont similaires. Par contre, passer d'un nombre à son opposé est plus compliqué. Cette notation possède une seule représentation de la valeur 0 ($v_i = 0,\ \forall\ i$). Cependant, l'ensemble des nombres représentables n'est plus symétrique autour de 0, il y a une valeur négative de plus que le nombre de valeurs positives.

– *La notation à retenue conservée* (ou *Carry Save*) est surtout utilisée dans les calculs intermédiaires. Il s'agit d'un système de représentation redondant [13], qui utilise $2 \times n$ bits notés $\{c_{n-1}, \cdots, c_0\}$ et $\{s_{n-1}, \cdots, s_0\}$. On travaille toujours en base 2, mais on a un ensemble de chiffres $\{0, 1, 2\}$, en posant $v_i = c_i + s_i$ pour chaque chiffre de la valeur $v = c + s$. Les nombres c et s peuvent être en non signé ou en complément à 2. Par exemple, l'extension à retenue conservée du complément à 2 est :

$$v = \text{ULP}(v) \left(-2^{n-1}\ (c_{n-1} + s_{n-1}) + \sum_{i=0}^{n-2} (c_i + s_i)\ 2^i \right),$$

$$\text{d'où } v \in \left([-2^n, 2^n - 2] \cap \mathbb{N} \right) \times \text{ULP}(v).$$

Cette notation présente l'avantage suivant : il est possible d'additionner *en parallèle* (ou *en temps constant*) un nombre en retenue conservée et un nombre en notation simple de position (ou un autre nombre en retenue conservée), et d'obtenir leur somme en retenue conservée. Par contre, la notation à retenue conservée utilise deux fois plus de bits que les notations simples de position, ce qui implique un routage et une mémorisation deux fois plus coûteux, et crée des opérateurs potentiellement plus gros.

– *La notation Borrow Save* est une autre notation binaire redondante de base 2 proche de la notation à retenue conservée, mais avec des chiffres dans l'ensemble $\{-1, 0, +1\}$. On code cette fois un chiffre v_i par soustraction de deux bits :

$v_i = v_i^+ - v_i^-$. Par la suite, pour éviter les confusions entre les chiffres négatifs et la soustraction, on note les chiffres négatifs en les surmontant d'une barre. Par exemple, $-5 = \overline{5}$. L'extension borrow-save du non signé est :

$$v = \text{ULP}(v) \sum_{i=0}^{n-1} (v_i^+ - v_i^-)\ 2^i,$$

$$\text{d'où } v \in \left([-2^n + 1, 2^n - 1] \cap \mathbb{N}\right) \times \text{ULP}(v).$$

Un des intérêts de cette notation par rapport à celle à retenue conservée est la manipulation plus naturelle des nombres négatifs. En particulier, calculer l'opposé d'un nombre est trivial, puisqu'il suffit de permuter les bits v_i^+ et v_i^-.

Le format virgule fixe est celui qui est le plus couramment utilisé dans les applications du traitement du signal que visent nos opérateurs. Dans le cas des applications de traitement des images, on travaille le plus souvent en arithmétique entière.

Un autre format, plus courant dans les processeurs généralistes, est le format virgule flottante. Il permet de représenter des valeurs avec une plus grande dynamique, au prix d'opérateurs plus complexes. Nous n'utiliserons pas ce format.

Exemple. Soit v codée par les 4 bits $(v_3, v_2, v_1, v_0) = (1, 1, 0, 1)$. La valeur v dépend de la notation :
- Si on veut coder des valeurs entières positives, le chiffre de poids faible v_0 doit avoir un poids de 1. On pose donc $\text{ULP}(v) = 1$, et on lit $v = 13$.
- Par contre, si on travaille dans l'intervalle $[0, 1[$, on pose $\text{ULP}(v) = 2^{-4}$ pour un codage non signé, et on lit $v = 13 \times 2^{-4} = \frac{13}{16} = 0.8125$.
- Si on travaille dans l'intervalle $[-1, 1[$ en complément à 2, on pose $\text{ULP}(v) = 2^{-4+1}$ (on utilise un fil pour le signe). On lit alors $v = (-2^3 + 5) \times 2^{-3} = -\frac{3}{8} = -0.375$.

Types de fonctions

On peut classer les fonctions en deux catégories [15] que sont les fonctions élémentaires (fonctions habituellement utilisées et qu'on peut construire facilement) et les fonctions spéciales (fonctions régulières usitées en physique qui ne sont pas élémentaires : erf, fonctions elliptiques, fonctions de Jacobi, etc.) qui sont difficilement évaluables. Les fonctions élémentaires sont elles-mêmes subdivisées entre fonctions algébriques (solutions d'une équation algébrique) et fonctions transcendantes (sin, tan, exp, log, etc.).

Ces catégories s'entendent pour les fonctions d'une variable (plus couramment utilisées) mais elles s'étendent facilement aux fonctions à plusieurs variables comme $\sqrt{x + y}$ qui est une fonction algébrique à deux variables.

Types d'évaluation

L'évaluation en matériel de fonctions numériques est un point crucial dans de nombreuses applications. L'inverse et la fonction sinus sont souvent utilisées en traitement du signal. Les fonctions transcendantes sont récurrentes en calcul flottant, et des fonctions algébriques comme la racine carrée et la norme euclidienne se retrouvent souvent dans des applications graphiques. Le calcul de ces fonctions dans des opérateurs spécialisés est souvent beaucoup plus efficace qu'avec des structures génériques complètes (additions et multiplications de taille inutilement importante). De nombreuses méthodes existent pour l'évaluation de fonctions en matériel : méthodes à base de tables, algorithmes à récurrence de chiffres, approximations polynomiales ou rationnelles, ou encore des combinaisons de ces méthodes [13, 15].

Les méthodes à base de tables sont souvent utilisées pour des applications avec de petits besoins en précision. Des précisions plus grandes peuvent être obtenues en combinant tables et opérations arithmétiques [20, 21].

Les algorithmes à récurrence de chiffres, ou algorithmes décalage-addition, produisent un chiffre du résultat par itération en partant du chiffre de poids le plus fort, comme la division à la main. Les deux plus fameux algorithmes à récurrence de chiffres sont : les algorithmes de type SRT pour la division, racine carrée et autres fonctions algébriques [13], et l'algorithme CORDIC pour certaines fonctions élémentaires [15]. Ces algorithmes utilisent seulement des additions et décalages (plus éventuellement des petites tables) pour le calcul de chaque itération. Ils donnent lieu à des opérateurs qui sont petits, mais qui ont une forte latence à cause de leur convergence linéaire. Augmenter leur base de travail réduit le nombre d'itérations nécessaires, mais demande des circuits sensiblement plus gros pour le calcul des itérations.

Les approximations polynomiales sont largement utilisées pour l'évaluation de fonctions, même en matériel où la taille des multiplieurs utilisés pour leur évaluation est un problème majeur. Ces approximations permettent d'évaluer la plupart des fonctions courantes pour peu qu'elles soient régulières. Les approximations rationnelles permettent une évaluation plus précise, mais elles requièrent une division qui s'ajoute au coût matériel si on la calcule de manière classique (un algorithme comme la E-méthode [17, 16] calcule cette division mais de manière « cachée » sans payer le prix d'une opération supplémentaire) et introduit une latence supplémentaire souvent importante.

Implantation matérielle

Le but d'une implantation matérielle des fonctions numériques est l'amélioration des performances. Par exemple, l'implantation de fonctions cryptographiques en matériel fait habituellement gagner plusieurs ordres de grandeur en vitesse par rapport aux mêmes fonctions codées en logiciel. En plus d'être rapide, un opérateur doit aussi occuper la plus petite surface possible pour des raisons de coût.

Maintenir une bibliothèque contenant les descriptions en VHDL (*Very-high-speed-integrated-circuit Hardware Description Language*) d'opérateurs spécialisés est difficile, du fait de la gestion limitée des paramètres génériques dans ce langage. Notre approche est donc, plutôt que de fournir une bibliothèque VHDL, d'écrire des programmes en C ou C++ qui génèrent des descriptions d'opérateurs optimisés. L'utilisation de générateurs permet d'effectuer simplement des calculs plus complexes que ceux qu'on ferait à l'aide de formules VHDL, et d'explorer l'espace des paramètres. On trouve ainsi des solutions bien plus efficaces que ce qu'on peut obtenir à l'aide du VHDL seul. La rigidité du VHDL rend aussi difficile l'écriture de certaines structures, même si elles sont régulières, comme par exemple des arbres.

Les implantations matérielles des opérateurs décrits dans cette thèse ont été faites sur des circuits de type FPGA (pour *Field-Programmable Gate Array*). Bien qu'ils soient moins performants que des circuits numériques spécialisés de type ASIC (pour *Application-Specific Integrated Circuits*), les FPGA ont un coût de développement bien moindre et leur mise en œuvre pour le prototypage est beaucoup plus facile et rapide. De plus, ils disposent de ressources suffisantes même pour des architectures complexes. Toutes ces raisons en font de bons candidats pour la mise en pratique de nos travaux.

Les résultats présentés ici peuvent être facilement adaptés aux circuits ASIC, moyennant quelques modifications pour adapter les opérateurs à la structure particulière des ASIC. Par exemple, l'absence de lignes de *fast-carry* demanderait l'utilisation d'un algorithme plus efficace que l'addition à propagation de retenue, comme des additionneurs à saut de retenue (*carry-skip*) [13].

Travaux

`Divgen` et la division

La division est à elle seule un sujet qui concentre de multiples questions. La représentation des chiffres et des nombres, le type d'algorithme sont autant de choix à faire dans la conception d'un opérateur de division. Cette conception n'est pas simple et le moindre changement de structure (pour une simple raison d'envie d'essayer d'autres paramètres) oblige à tout repenser, tout réécrire, et à devoir revalider l'opérateur.

Pour cette raison, nous avons décidé de développer un générateur de diviseurs. Ce générateur prend simplement en entrée un fichier décrivant tous les paramètres choisis (voir la figure 1) et il donne une description en langage VHDL synthétisable et optimisé du circuit correspondant. Ce projet a été utilisé lors d'une collaboration avec le laboratoire LETI du CEA.

Il a donné suite à la publication d'un logiciel sous licence GPL [5]. Il a aussi permis une approche et une découverte des générateurs automatiques d'opérateurs arithmétiques.

On peut choisir par exemple la taille et la représentation du dividende, du diviseur, du résultat, l'algorithme utilisé, la représentation des signaux internes, le type de table de sélection utilisé, la génération d'une figure pour représenter cette table, etc.

La figure 1 représente une configuration possible. Ici les opérandes seront toutes représentées en non-signé, le dividende sur 9 bits, le diviseur sur 6 bits et le quotient sur 7. On utilisera un algorithme SRT, etc.

La fonction de sélection, qui a pour but de choisir à chaque étape le chiffre suivant du quotient, est un point crucial de ce genre d'opérateurs et elle nécessite un soin particulier à la conception. Des outils comme `Divgen` permettent de tester différentes options, d'explorer l'espace des paramètres et d'en connaître les conséquences avant de faire un choix d'implantation. On peut voir sur la figure 2 un exemple de comparaison du temps total nécessaire à différents algorithmes. Tous les opérateurs considérés ont été générés par `Divgen` et synthétisés sur un FPGA de type Virtex-E de Xilinx. Ce genre de comparaisons (comme celles des figures 3 et 4) est réalisable très facilement et très rapidement grâce à `Divgen`.

Ces travaux ont été présentés à la conférence SPIE à San Diego en 2005 [5].

```
1  x_representation unsigned
2  d_representation unsigned
3  q_representation unsigned
4  x_size 9
5  d_size 6
6  q_size 7
7  algorithm SRT
8  q_radix 4
9  q_max_digit 3
10 partial_remainder_representation 2s_complement
11 step_adder RCA
12 #guard_bits 0
13 SRT_table_folding no
14 gray_encoding no
15 SRT_table_fig n&b
```

Figure 1 *Exemple de fichier de configuration pour* `Divgen`.

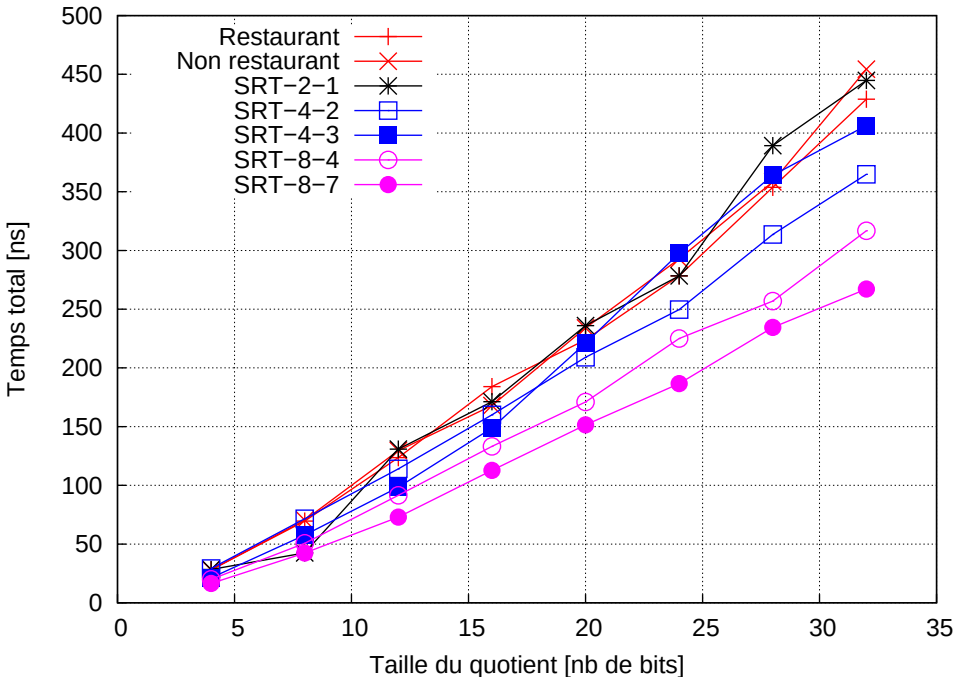


Figure 2 *Temps total de la division pour différentes tailles et différents algorithmes. Opérateurs générés par* `Divgen` *et synthétisés sur un FPGA de type Virtex-E de Xilinx.*

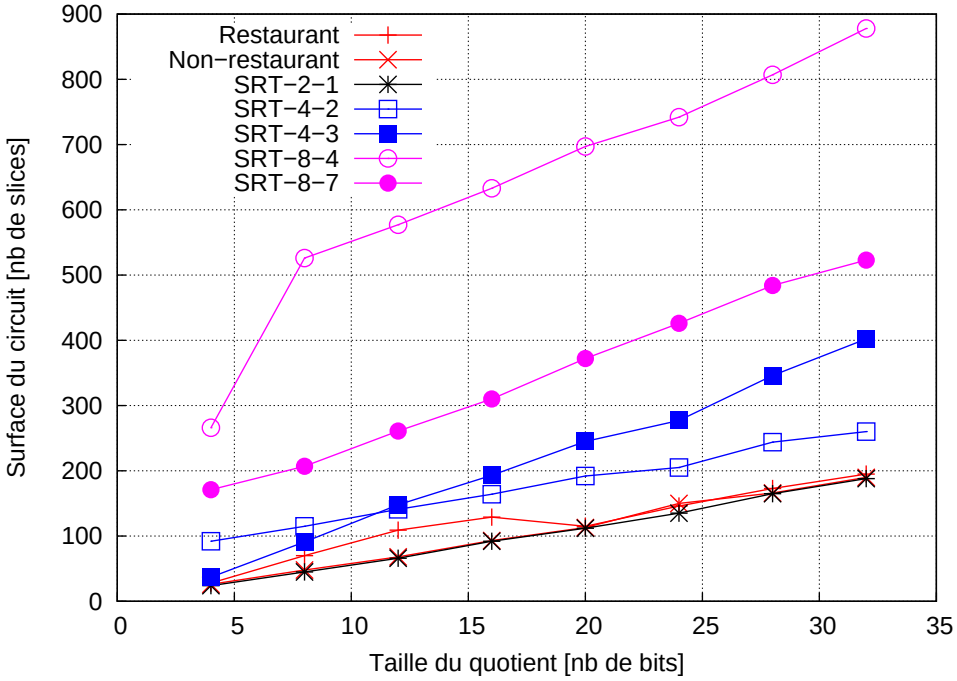


Figure 3 *Comparaison de la surface des opérateurs.*

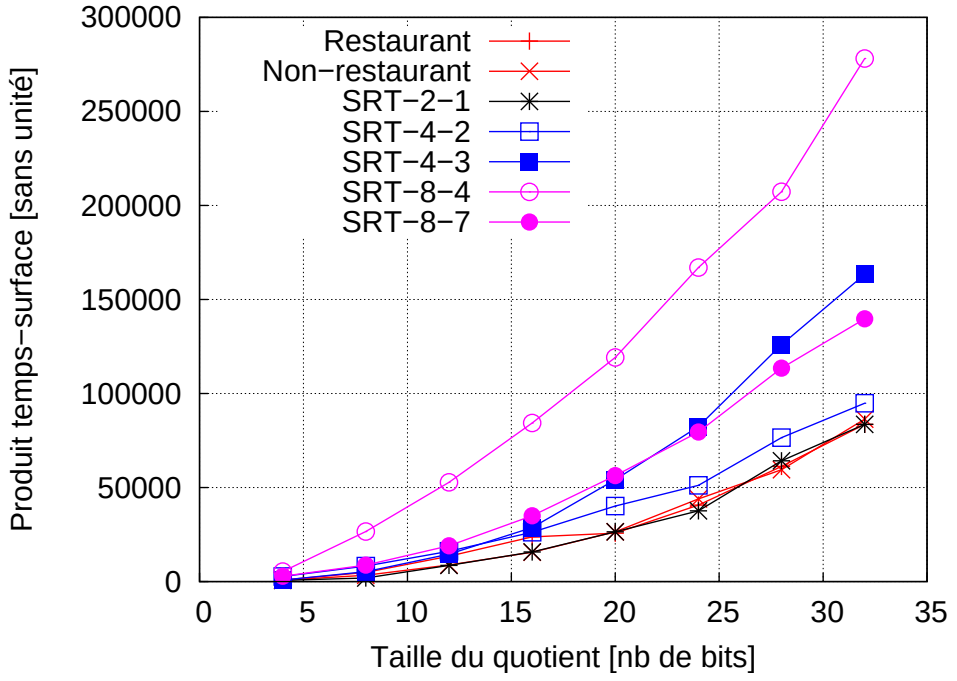


Figure 4 *Comparaison du produit temps× surface.*

E-méthode

La E-méthode a été introduite par M. Ercegovac dans les années 70 [17, 16]. Elle permet l'évaluation de certaines fonctions par approximations polynomiales ou rationnelles en résolvant un système linéaire à diagonale dominante, à l'aide d'une itération simple et régulière à base d'additions et de décalages. Les systèmes linéaires cibles sont de la forme :

$$\begin{pmatrix} 1 & -x & 0 & & \cdots & & & 0 \\ q_1 & 1 & -x & 0 & & \cdots & & 0 \\ q_2 & 0 & 1 & -x & 0 & & \cdots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & & \vdots \\ & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & & & \ddots & \ddots & \ddots & 0 \\ & & & & & \ddots & 1 & -x \\ q_n & 0 & & \cdots & & & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ \vdots \\ \vdots \\ p_{n-1} \\ p_n \end{pmatrix} \quad (1)$$

On note $\mathcal{A}$ la matrice de ce système, b le vecteur second membre et y le vecteur solution. La taille du système est $n+1$.

Après résolution du système 1, la première composante du vecteur solution y est la valeur au point x de la fraction rationnelle R, de degré n, dont les coefficients du numérateur sont les composantes de b et ceux du dénominateur sont les composantes de la première colonne de $\mathcal{A}$. C'est à dire que la solution de $\mathcal{A}y = b$ est $y = [y_0, y_1, \ldots, y_n]^t$ telle que

$$y_0 = R(x) = \frac{P(x)}{Q(x)} = \frac{p_n x^n + p_{n-1}x^{n-1} + \cdots + p_0}{q_n x^n + q_{n-1}x^{n-1} + \cdots + 1}.$$

La E-méthode permet donc d'évaluer des fractions rationnelles en un point.

En simplifiant légèrement la matrice $\mathcal{A}$, la E-méthode permet aussi d'évaluer des polynômes. En effet, si tous les q_i sont nuls (sauf $q_0 = 1$), alors la première composante du vecteur solution est la valeur au point x du polynôme P, de degré n, dont les coefficients sont les composantes de b. La solution de $\mathcal{A}y = b$ est alors $y = [y_0, y_1, \ldots, y_n]^t$ avec

$$y_0 = P(x) = p_n x^n + p_{n-1}x^{n-1} + \cdots + p_0.$$

Introduisons quelques notations utiles pour la suite. La base du système de représentation des nombres est notée β (en pratique, $\beta \in \{2, 4, 8\}$ dans ce travail). Nous abandonnons la simplification $\beta = 2$ introduite précédemment au profit d'une généralisation de cette base. Le vecteur des restes partiels, de taille $n+1$, est noté w. Les différentes valeurs d'une quantité dans le temps sont représentées avec la notation crochet (comme en traitement du signal). Par exemple $w[j]$ dénote le vecteur des restes

partiels à la $j$ième itération. Le vecteur de chiffres du résultat trouvé à chaque itération j est noté $d[j]$.

L'algorithme de E-méthode est présenté en figure 5. Le vecteur des restes partiels est initialisé avec les coefficients du polynôme P (les composantes de b). Le premier vecteur de chiffres du résultat est le vecteur nul.

initialisation :
  $w[0] \longleftarrow b$
  $d[0] \longleftarrow 0$
itération :
  pour j de 1 à m faire
    $w[j] \longleftarrow \beta \times \Big(w[j-1] - \mathcal{A} \times d[j-1] \Big)$
    $d[j] \longleftarrow S(w[j])$
résultat :
    $y_0[m] = \sum_{i=1}^{m} d_0[i]\beta^{-i}$

Figure 5 *Algorithme d'évaluation avec la E-méthode (version vectorielle).*

Comme tous les algorithmes à base d'addition et de décalages, la E-méthode produit un chiffre du résultat à chaque itération. Comme les algorithmes de division, il procède en commençant par les bits de poids fort. La concaténation des différents chiffres fournit une valeur qui tend vers la valeur mathématique du résultat (à l'infini). Ici, le résultat de chaque itération est un vecteur de chiffres $d[j]$. L'itération est basée sur un calcul similaire à celui d'une division où l'on « diviserait » par la matrice $\mathcal{A}$ (d'où le terme reste partiel pour w). Pour chaque ligne $i = 1, \ldots, n-1$ de la matrice $\mathcal{A}$, le calcul effectué est :

$$w_i[j] = \beta \times \big(w_i[j-1] - d_0[j-1]q_i - d_i[j-1] + d_{i+1}[j-1]x \big) \qquad (2)$$

Dans les cas $i = 0$ et $i = n$, le calcul se simplifie en $w_0[j] = \beta \times \big(w_0[j-1] - d_0[j-1] + d_1[j-1]x \big)$ et $w_n[j] = \beta \times \big(w_n[j-1] - d_0[j-1]q_n - d_n[j-1] \big)$.

Le calcul des nouveaux termes du reste partiel n'implique que des additions/soustractions et des produits d'un nombre par un seul chiffre.

A chaque itération, un nouveau vecteur de chiffres du résultat $d[j]$ est produit. Ce calcul se fait en utilisant une fonction de sélection très simple, calculée au moyen d'une table qui examine les bits de poids fort du reste partiel.

La figure 6 donne une illustration de l'architecture matérielle employée pour le calcul d'une ligne du système à chaque itération dans les deux cas que sont l'approximation polynomiale et l'approximation rationnelle.

Cet algorithme a conduit à deux études. La première est consacrée à l'implantation d'opérateurs en grande base [7] et a été présentée à la conférence SYMPA, au Croisic, en 2005. Cet article montre que les gains en vitesse sont substantiels pour un coût en surface de circuit relativement limité. L'utilisation de grandes bases peut donc s'avérer

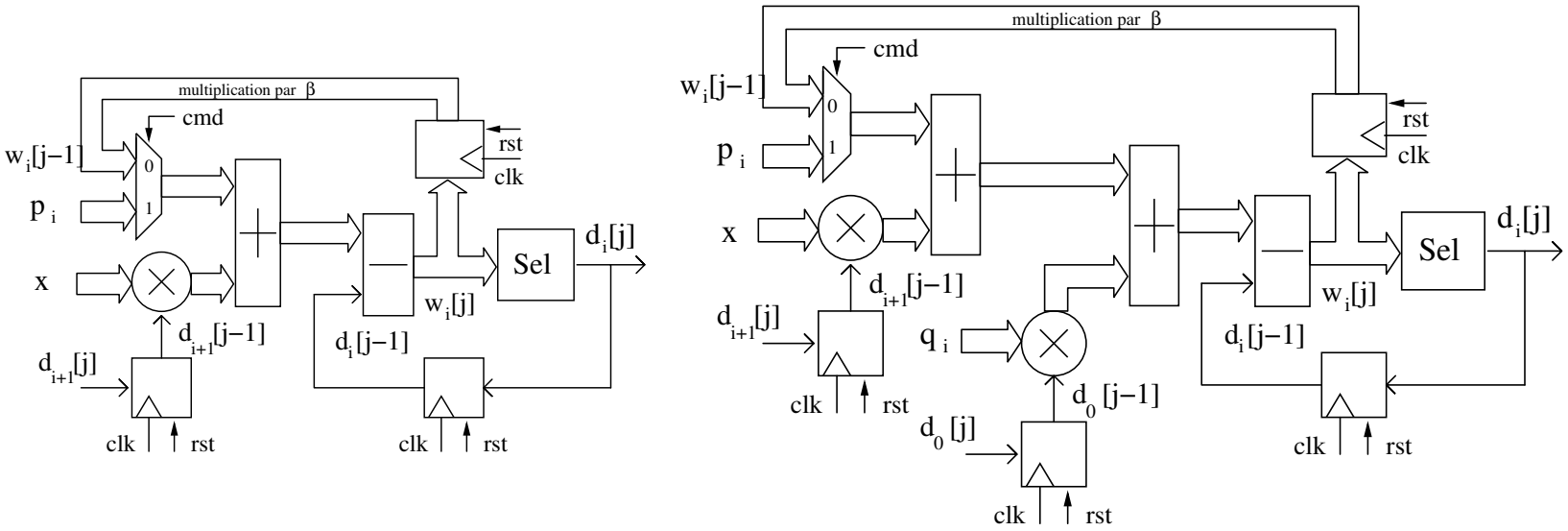


Figure 6 *Unité fonctionnelle de calcul d'une ligne de l'itération à l'étape j dans le cas polynomial (à gauche) et dans le cas rationnel (à droite).*

utile en fonction de la performance recherchée. Les courbes de la figure 7 illustrent le temps de calcul total pour chacune des solutions polynomiales et chacune des solutions rationnelles. Il est clair sur cette figure que la base 8 permet d'obtenir des temps de calcul totaux bien plus faibles. Cette augmentation de la base se paye en termes de surface.

La seconde partie des travaux concernant cet algorithme réalise une étude statistique de la fonction de sélection en vue de diminuer la consommation électrique [6] et a été présentée à la conférence FTFC, à Paris, en 2005. En effet la fonction de sélection permet, si on utilise une notation redondante des chiffres, une certaine latitude dans le choix du nouveau chiffre. Dans ce cas on essaie de choisir le chiffre précédemment sélectionné, si ce choix est acceptable. La figure 8 représente la modification que subit l'architecture matérielle de l'opérateur.

Cela a pour effet de limiter l'activité (donc la consommation électrique) de l'addition du calcul de $w[j] \longleftarrow \beta \times \Big(w[j-1] - \mathcal{A} \times d[j-1] \Big)$.

La table 1 présente les statistiques du choix du chiffre du résultat à l'itération j en fonction de sa valeur à l'itération $j-1$ pour une fonction de sélection classique (sans mémoire). La table 2 présente les mêmes statistiques obtenues pour les mêmes paramètres et valeurs en utilisant la fonction de sélection à mémoire. On constate bien que le nombre de cas où le chiffre du résultat choisi est k à l'itération j alors qu'il valait déjà k à l'itération $j-1$ est augmenté (ce sont les cases de la diagonale).

L'activité est alors réduite d'environ 30%, ce qui permet d'envisager une réduction substantielle de la consommation électrique.

		$d_i[j-1]$						
		-3	-2	-1	0	1	2	3
$d_i[j]$	-3	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	-2	0.01	0.02	0.01	0.02	0.03	0.03	0.01
	-1	0.00	0.01	0.10	0.34	0.25	0.02	0.00
	0	0.00	0.01	0.02	2.60	0.45	0.01	0.00
	1	0.03	0.02	0.26	0.58	0.25	0.00	0.01
	2	0.00	0.01	0.04	0.04	0.02	0.00	0.00
	3	0.00	0.01	0.00	0.00	0.00	0.01	0.00

Table 1 *Statistiques de la sélection standard.*

Il est vrai que cet algorithme n'est pas tout récent mais il suscite toutefois un intérêt certain dans la recherche actuelle [18].

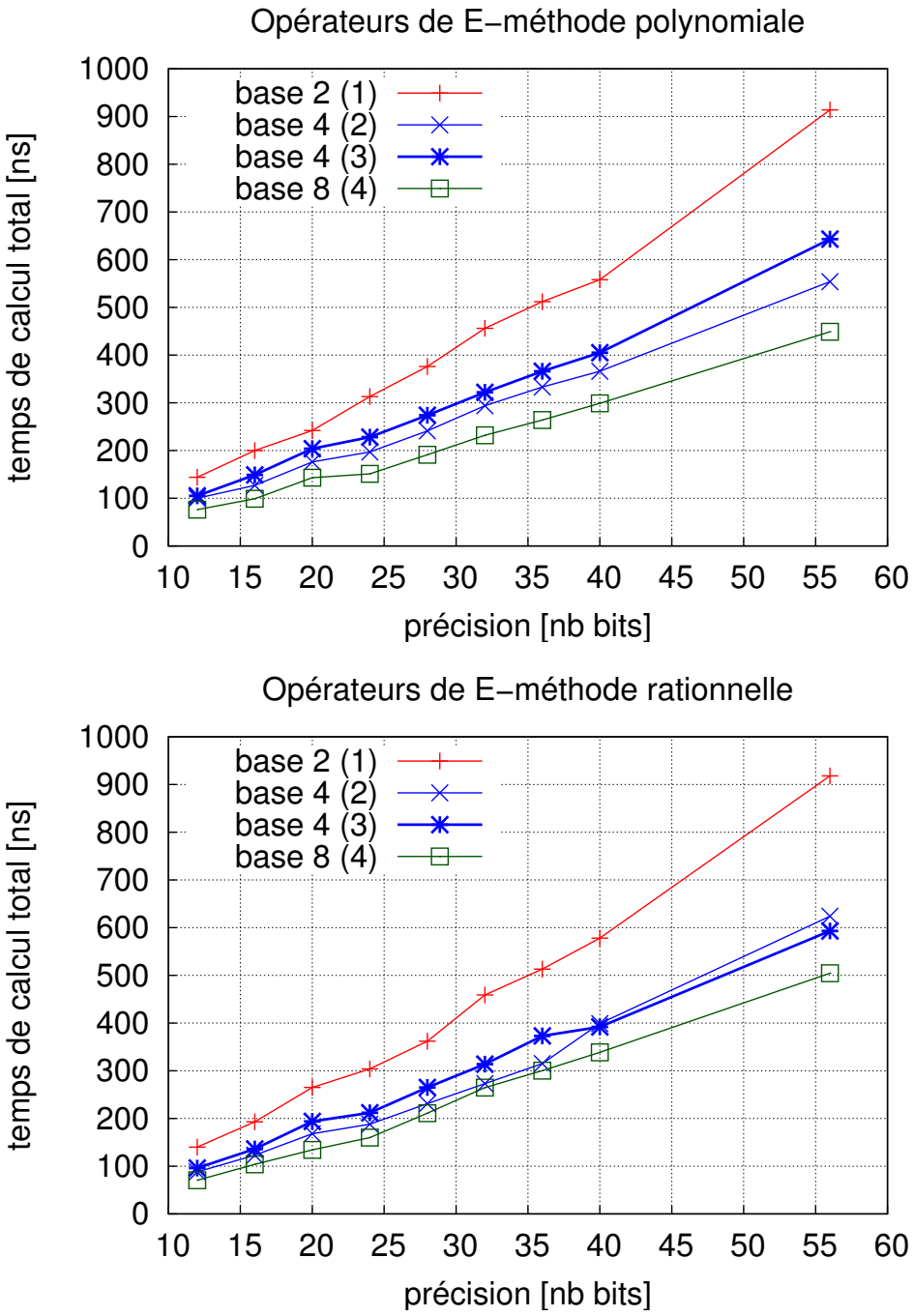


Figure 7 *Temps de calcul total pour les différentes solutions en fonction de la base (solution polynomiale en haut et rationnelle en bas).*

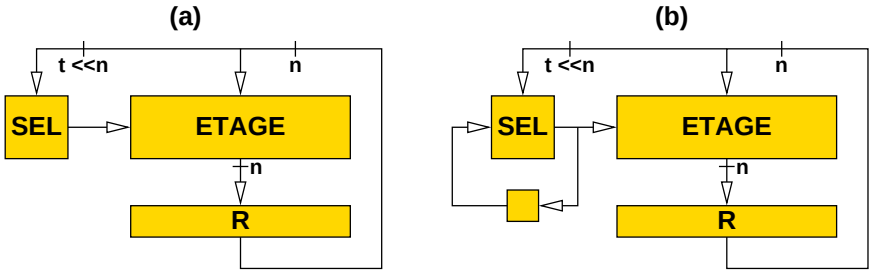


Figure 8 *Étage de calcul et fonction de sélection sans (a) et avec (b) mémoire.*

		$d_i[j-1]$						
		-3	-2	-1	0	1	2	3
$d_i[j]$	-3	0.00	0.00	0.00	0.02	0.01	0.00	0.00
	-2	0.00	0.01	0.00	0.01	0.07	0.00	0.00
	-1	0.00	0.00	0.02	0.03	0.02	0.02	0.02
	0	0.02	0.00	0.02	3.00	0.31	0.01	0.07
	1	0.00	0.00	0.00	0.53	0.49	0.01	0.03
	2	0.00	0.00	0.03	0.17	0.03	0.14	0.01
	3	0.00	0.00	0.02	0.08	0.01	0.02	0.01

Table 2 *Statistiques de la sélection avec mémoire.*

Évaluation rapide de fonctions

L'évaluation de fonctions reste un domaine très actif car beaucoup de champs applicatifs la requièrent. Comme il a été dit précédemment, l'utilisation d'opérateurs matériels spécialisés est souvent préférée à l'utilisation d'une évaluation en logiciel sur un processeur générique.

Compte tenu de l'impossibilité de faire des calculs en précision infinie, l'évaluation est approchée. Il existe donc une erreur ϵ entre la valeur réelle de la fonction et la valeur estimée. On peut alors se préoccuper de minimiser la valeur moyenne de cette erreur sur l'intervalle considéré, ϵ_{moy}, ou sa valeur maximale, ϵ_{max}. La philosophie qui sous-tend la conception d'un opérateur est alors différente. Nous avons travaillé sur ces deux approches.

- **Erreur moyenne**
  Nous avons eu l'idée de proposer une façon de générer des opérateurs arithmétiques qui soient *petits*. Les opérateurs utilisés en traitement du signal, traitement des images ou en contrôle numérique ont rarement besoin d'une grande précision et nous avons eu l'idée de profiter de ce besoin limité en précision pour réduire la taille (donc le coût, le temps d'exécution et la consommation électrique) de ces opérateurs.
  La figure 9 illustre parfaitement le souci que nous avons eu de limiter chaque

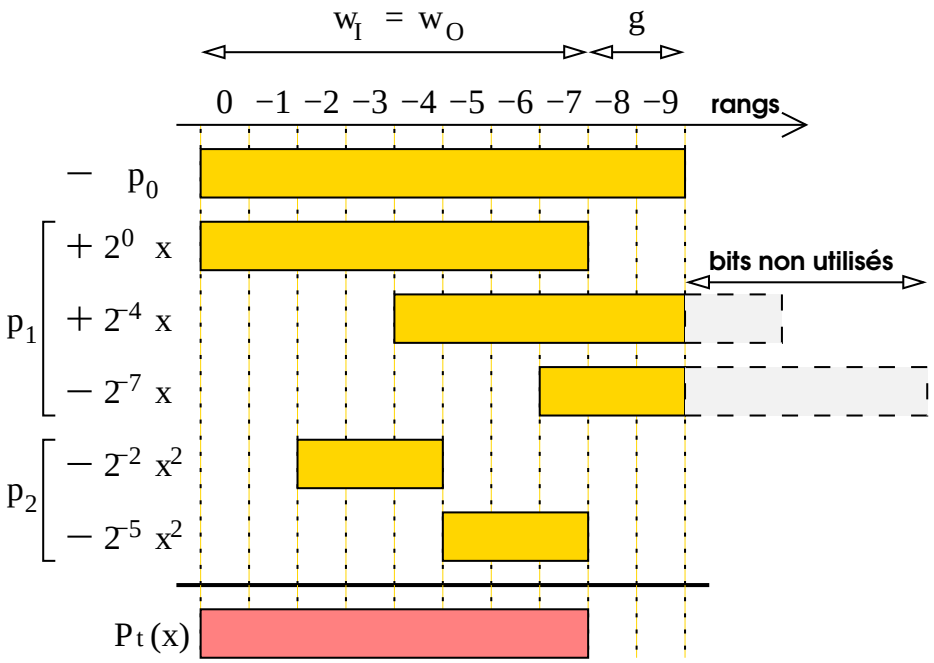


Figure 9 *Illustration du calcul du polynôme d'approximation de* $sin(x)$ *sur* $[0, \pi/4[$.

étape du calcul au strict nécessaire. Il est inutile de procéder à des calculs intermédiaires qui apporteront très peu d'informations et beaucoup d'inconvénients. Il est toutefois utile de réaliser ces calculs intermédiaires avec une précision suffisante pour ne pas rendre le résultat inutilisable (en pratique nous avons très souvent utilisé 2 bits de garde, $g = 2$).
Nous estimons qu'un calcul très précis quant à l'évaluation de la fonction concernée est superflu. Nous approchons donc cette fonction par un polynôme avec la contrainte de faire disparaître toute multiplication (opération coûteuse) dans son évaluation. Nous obtenons ce résultat en faisant une estimation des puissances de l'argument, et en contraignant les coefficients du dit polynôme. Ce dernier point a suscité un travail considérable auprès d'autres chercheurs.
La table 3 met en évidence les gains obtenus à chaque étape de notre optimisation au prix d'une perte de précision qui reste très supportable. Nous passons de trois multiplications et 2 additions/soustractions à 6 additions/soustractions. C'est un gain substantiel.
Nous avons réalisé une première étude très prometteuse [4] qui a obtenu le *prix du meilleur papier* à la conférence ASAP05 et qui permet d'obtenir des opérateurs jusqu'à 42% plus petits que les meilleurs opérateurs du moment. Cette étude a ensuite conduit à d'autres travaux [3, 1] présentés aux conférences SPIE, à San Diego et SiPS, à Banff (Canada), toutes les deux en 2006.

- **Erreur maximale**
  La conception d'opérateurs arithmétiques demande un soin particulier à chaque

Étape	ϵ_{moy}	ϵ_{max}	Coût
Minimax	$\epsilon_{th} = 0.23 \times 10^{-2}$		$3\times$, $2\pm$
Quantif.	0.16×10^{-2}	0.53×10^{-2}	$1\times$, $2\pm$
Estim. de x^2	0.69×10^{-2}	0.23×10^{-1}	$7\pm$
Optimisation	0.41×10^{-2}	0.18×10^{-1}	$6\pm$

Table 3 *Évolution de la précision et du coût pour les différentes étapes du calcul de* $sin(x)$ *sur* $[0, \pi/4[$ *par une approximation du second degré.*

étape. Il est assez facile de savoir quelle doit être la taille des signaux en entrée ou en sortie de l'opérateur, elles dépendent souvent de l'environnement dans lequel l'opérateur sera amené à travailler. En revanche, la taille des signaux intermédiaires, nécessaires aux différentes étapes du calcul, est beaucoup plus difficile à déterminer alors qu'elle influence très directement toutes les performances du circuit. Une partie de nos travaux a consisté à développer un moyen simple pour déterminer la taille optimale de ces signaux. Cette méthode s'appuie sur l'outil `GAPPA` [19] pour borner finement les erreurs commises pendant l'évaluation et déterminer *a priori* la précision finale de l'opérateur.
Ces travaux ont été présentés à la conférence SYMPA, à Perpignan, en 2006 [2]. Ils ont été étendus dans un article, à paraître en 2008, du journal *Technique et Science Informatique* [8]. L'utilisation d'une telle méthode nous a permis d'obtenir des opérateurs jusqu'à 40% plus petits et 51% plus rapides qu'avec une conception basique.

Conclusion et perspectives

Conclusion

Les différents travaux réalisés dans le cadre de cette thèse ont permis d'obtenir des implantations efficaces d'opérateurs arithmétiques optimisés sur FPGA. Des générateurs automatiques ont été développés qui servent à écrire et valider facilement des opérateurs adaptés à de nombreuses cibles et utilisations.

Le logiciel `Divgen`, présenté lors de la conférence internationale *SPIE - Advanced Signal Processing Algorithms, Architectures and Implementations XV* à San Diego en 2005 [5], permet d'obtenir facilement les descriptions VHDL de diviseurs optimisés. Ces diviseurs sont générés pour différents algorithmes et options passés en paramètres. Il s'agit d'un programme C++ de 5000 lignes environ. Une extension vers d'autres fonctions algébriques est implantée (en partie) par un programme Maple de 2000 lignes environ. Ce programme permet d'obtenir des opérateurs à récurrence de chiffres très efficaces grâce à un calcul fin des bornes d'erreurs. En effet, il peut manipuler des termes qu'il est difficile de gérer lors d'une démonstration manuelle. De plus, une étude complète faite à la main pour un jeu de paramètres donné est souvent fastidieuse et sujette à de nombreuses erreurs. La génération automatique de descriptions VHDL d'opérateurs à additions et décalages optimisés et validés numériquement fait donc de ces opérateurs une solution intéressante pour des non spécialistes.

La E-méthode est un algorithme introduit à la fin des années 70. Il permet d'évaluer une fraction rationnelle (ou un polynôme, évidemment) en un point. L'évaluation ne répond pas aux schémas classiques utilisés pour cette opération, elle correspond au calcul de la solution d'un système linéaire relativement simple. Nous avons conduit deux études sur ce sujet. La première a été publiée au *Symposium en Architecture de machines*, au Croisic en 2005 [7]. Elle traite de l'implantation de cet algorithme en grande base et des gains apportés. La seconde a été publiée à la conférence *Faible Tension Faible Consommation*, à Paris en 2005 [6]. Elle explore les possibilités offertes par l'utilisation d'une représentation redondante des chiffres lors de l'emploi de cet algorithme, au profit d'une baisse de la consommation électrique. Ces résultats sont prometteurs mais seulement théoriques pour le moment. Ils mériteraient d'être exploités de manière pratique.

Nous avons travaillé sur des opérateurs évaluant un ensemble de fonctions à l'aide d'approximations polynomiales. Un algorithme implanté dans un programme C++ de

plus de 5000 lignes fournit des approximations avec des coefficients creux et des puissances tronquées pour minimiser leur coût matériel. Un grande partie du matériel est partagée entre l'évaluation des différentes fonctions, ce qui permet d'en faire une solution bien plus intéressante qu'une implantation séparée des opérateurs. Le choix du langage C++ et une optimisation poussée de notre programme rendent possible une exploration vaste de l'espace des paramètres. La version initiale de cette méthode a été présentée lors de la *16ème International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, à Samos en 2005 [4]. Elle y a remporté le *prix du meilleur papier*. La méthode de génération des opérateurs spécialisés dans le calcul des puissances utilisés par nos approximations a été présentée lors de la conférence internationale *SPIE - Advanced Signal Processing Algorithms, Architectures and Implementations XVI*, à San Diego en 2006 [3].

L'évaluation de fonctions a aussi été envisagée sous l'angle de la minimisation de l'erreur maximale commise sur un intervalle. Dans ce but, un soin particulier a été apporté à la détermination de la taille des signaux intermédiaires nécessaires aux différentes étapes du calcul. Elle est très difficile à déterminer alors qu'elle a une influence sur toutes les performances du circuit. Ces travaux ont été présentés au *Symposium en Architecture de machines*, à Perpignan en 2006 [2]. Une version étendue de ces travaux a été acceptée pour publication dans le journal *Technique et Science Informatique* en 2008 [8].

Perspectives

L'algorithme de E-méthode a initié des opportunités du point de vue de la réduction de la consommation énergétique. Les travaux sur ce sujet sont, pour l'instant, uniquement théoriques et statistiques, ils seront exploités de manière pratique par la suite.

Une comparaison de la consommation électrique de différentes familles d'opérateurs arithmétiques est en cours. Elle sera publiée prochainement pour apporter aux concepteurs des connaissances sur cette question qui leur permettront de faire un choix parmi les diverses possibilités d'implantation qui s'offrent à eux.

Je compte élargir l'éventail des algorithmes concernés par ces travaux. Des algorithmes comme CORDIC ou ceux à destination de la virgule flottante n'ont pas encore attiré suffisamment mon attention, ils seront étudiés car ils sont d'un intérêt certain dans l'arithmétique des ordinateurs d'aujourd'hui.

De la même manière, seuls les opérateurs à destination de FPGA ont été ciblés. L'architecture ASIC est incontournable. Elle permettra sans doute des performances différentes, en particulier en ce qui concerne la consommation d'énergie. Ces cibles demandent la mise en place et la maîtrise d'une plate-forme de développement et d'outils logiciels précis et performants.

La consommation électrique est mon principal souci pour les travaux à venir et je souhaite me concentrer sur cette question. Elle est aujourd'hui une contrainte majeure dans un certain nombre de domaines (les systèmes embarqués, l'informatique

médicale,...) et nécessite d'être prise en compte très sérieusement au moment de la conception.

Publications personnelles

[1] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Carry prediction and selection for truncated multiplication. *Dans IEEE 2006 Workshop on Signal Processing Systems (SiPS'06)*, Banff Park Lodge, Banff, AB, Canada, octobre 2006. IEEE.

[2] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. *Dans 11ième SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 130–141, Perpignan, octobre 2006.

[3] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : New identities and transformations for hardware power operators. *Dans* F. T. LUK, éditeur : *Advanced Signal Processing Algorithms, Architectures and Implementations XVI*, San Diego, California, U.S.A., août 2006. SPIE.

[4] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Small FPGA polynomial approximations with 3-bit coefficients and low-precision estimations of the powers of x. *Dans* S. VASSILIADIS, N. DIMOPOULOS et S. RAJOPADHYE, éditeurs : *16th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 334–339, Samos, Greece, juillet 2005. IEEE Computer Society. Best Paper Award.

[5] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Divgen : a divider unit generator. *Dans* F. T. LUK, éditeur : *Advanced Signal Processing Algorithms, Architectures and Implementations XV*, volume 5910, page 59100M, San Diego, California, U.S.A., août 2005. SPIE.

[6] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Étude statistique de l'activité de la fonction de sélection dans l'algorithme de e-méthode. *Dans 5ième journées d'études Faible Tension Faible Consommation (FTFC)*, pages 61–65, Paris, mai 2005.

[7] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Évaluation de polynômes et de fractions rationnelles sur FPGA avec des opérateurs à additions et décalages en grande base. *Dans 10ième SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 85–96, Le Croisic, avril 2005.

[8] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. *Techniques et science informatiques (TSI)*, 2008. À paraître.

Références générales

[9] B. RANDELL : From analytical engine to electronic digital computer : The contributions of Ludgate, Torres and Bush. *IEEE Annals of the History of Computing*, 04(4):327–341, 1982.

[10] P. E. CERUZZI : The early computers of Konrad Zuse, 1935 to 1945. *Annals of the History of Computing*, 3(3):241–262, 1981.

[11] W. ASPRAY, A. G. BROMLEY, M. CAMPBELL-KELLY, P. E. CERUZZI et M. R. WILLIAMS : *Computing Before Computers*. Iowa State University Press, Ames, Iowa, 1990. Available at http://ed-thelen.org/comp-hist/CBC.html.

[12] Wikipédia, l'encyclopédie libre - loi de moore. http://fr.wikipedia.org/wiki/Loi_de_moore.

[13] M. D. ERCEGOVAC et T. LANG : *Digital Arithmetic*. Morgan Kaufmann, 2003.

[14] J.-M. MULLER : *Arithmétique des ordinateurs*. Masson, 1989.

[15] J.-M. MULLER : *Elementary Functions : Algorithms and Implementation*. Birkhäuser, 2ème édition, 2006.

[16] M. D. ERCEGOVAC : A general hardware-oriented method for evaluation of functions and computations in a digital computer. *IEEE Transactions on Computers*, C-26(7):667–680, 1977.

[17] M. D. ERCEGOVAC : *A general method for evaluation of functions and computation in a digital computer*. Thèse de doctorat, Dept. of Computer Science, University of Illinois, Urbana-Champaign, 1975.

[18] M. D. ERCEGOVAC et J.-M. MULLER : Solving systems of linear equations in complex domain : Complex e-method. LIP report No 2007-2, 2007. http://prunel.ccsd.cnrs.fr/ensl-00125369.

[19] G. MELQUIOND : Gappa : Génération automatique de preuves de propriétés arithmétiques. http://lipforge.ens-lyon.fr/www/gappa/.

[20] F. de DINECHIN et A. TISSERAND : Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, mars 2005.

[21] J. DETREY et F. de DINECHIN : Second order function approximation using a single multiplication on FPGAs. *Dans 14th International Conference on Field-Programmable Logic and Applications (FPL)*, numéro 3203 de LNCS, pages 221–230. Springer, septembre 2004.

[22] J.-M. MULLER : *Elementary Functions : Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[23] E. REMES : Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, 198:2063–2065, 1934.

[24] M.D. ERCEGOVAC, T. LANG, J.-M. MULLER et A. TISSERAND : Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, 49(7):627–637, juillet 2000.

[25] J. A. PINEIRO, J. D. BRUGUERA et J.-M. MULLER : Faithful powering computation using table look-up and a fused accumulation tree. *Dans Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 40–47. IEEE CS, juin 2001.

[26] N. TAKAGI : Powering by a table look-up and a multiplication with operand modification. *IEEE Transactions on Computers*, 47(11):1216–1222, 1998.

[27] D. A. SUNDERLAND, R. A. STRAUCH, S. S. WHARFIELD, H. T. PETERSON et C. R. ROLE : CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. *IEEE Journal of Solid State Circuit*, 19(4):497–506, août 1984.

[28] M. SCHULTE et J. STINE : Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, 48(8):842–847, août 1999.

[29] H. HASSLER et N. TAKAGI : Function evaluation by table look-up and addition. *Dans* S. KNOWLES et W.H. MCALLISTER, éditeurs : *12th IEEE Symposium on Computer Arithmetic*, pages 10–16. IEEE CS, juillet 1995.

[30] A. A. LIDDICOAT et M. J. FLYNN : Parallel square and cube computations. *Dans 34th Asilomar Conference on Signals, Systems, and Computers*, pages 1325–1329. IEEE, octobre 2000.

[31] J. DETREY et F. de DINECHIN : Second order function approximation using a single multiplication on FPGAs. *Dans 14th International Conference on Field-Programmable Logic and Applications*, pages 221–230. LNCS 3203, août 2004.

[32] N. BRISEBARRE et J.-M. MULLER : Functions approximable by e-fractions. *Dans 38th Conference on signals, systems and computers*, Pacific Grove, California, US, novembre 2004.

[33] N. BRISEBARRE, F. HENNECART, J.-M. MULLER, A. TISSERAND et S. TORRES. : MEPLib. http ://lipforge.ens-lyon.fr/, 2004.

[34] M.D. ERCEGOVAC, J.-M. MULLER et A. TISSERAND : FPGA implementation of polynomial evaluation algorithm. *Dans* SPIE, éditeur : *Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, volume 2607, pages 177–188, octobre 1995.

[35] J. VUILLEMIN : Exact real computer arithmetic with continued fractions. *Dans Proc. of the 1988 ACM conference on LISP and functional programming (LFP'88)*, pages 14–27. ACM Press, 1988.

[36] O. MENCER, M. MORF, A. LIDDICOAT et M. J. FLYNN : Efficient digit-serial rational function approximations and digital filtering applications. *Dans* IEEE, éditeur : *Asilomar Conference on Signals, Systems, and Computers*, novembre 1999.

[37] F. de DINECHIN et A. TISSERAND : Some improvements on multipartite tables methods. *Dans* N. BURGESS et L. CIMINIERA, éditeurs : *15th International Symposium on Computer Arithmetic ARITH15*, pages 128–135, Vail, Colorado, juin 2001. IEEE.

[38] M. D. ERCEGOVAC et T. LANG : *Division and Square-Root Algorithms : Digit-Recurrence Algorithms and Implementations.* Kluwer Academic, 1994.

[39] J. VOLDER : The CORDIC computing technique. *IRE Transactions on Computers*, EC-8(3):330–334, 1959.

[40] J. WALTHER : A unified algorithm for elementary functions. *Dans Joint Computer Conference Proceedings*, 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA,1990.

[41] W. CODY et W. WAITE : *Software manual for the elementary functions.* Prentice-Hall Inc, 1980.

[42] J.F. HART : *Computer Approximations.* Wiley, 1968.

[43] Y. C. LIM : Single-precision multiplier with reduced circuit complexity for signal processing applications. *IEEE Transactions on Computers*, 41(10):1333–1336, octobre 1992.

[44] M. J. SCHULTE et E. E. SWARTZLANDER : Truncated multiplication with correction constant. *Dans IEEE Workshop on VLSI Signal Proccessing VI*, pages 388–396, octobre 1993.

[45] E. J. KING et E. E. SWARTZLANDER : Data-dependent truncation scheme for parallel multipliers. *Dans Proc. of 31th Asilomar Conference on Signals, Systems & Computers*, volume 2, pages 1178–1182. IEEE, novembre 1997.

[46] E. E. SWARTZLANDER : Truncated multiplication with approximate rounding. *Dans Proc. of 33th Asilomar Conference on Signals, Systems & Computers*, volume 2, pages 1480–1483. IEEE, octobre 1999.

[47] E. G. WALTERS et M. J. SCHULTE : Efficient function approximation using truncated multipliers and squarers. *Dans Proc. of the 17th IEEE Symposium on Computer Arithmetic*, pages 232–239. IEEE Computer Society, juin 2005.

[48] J. E. STINE et O. M. DUVERNE : Variations on truncated multiplication. *Dans Proc. Euromicro Symposium on Digital System Design (DSD)*, pages 112–119. IEEE, septembre 2003.

[49] N. YOSHIDA, E. GOTO et S. ICHIKAWA : Pseudorandom rounding for truncated multipliers. *IEEE Transactions on Computers*, 40(9):1065–1067, septembre 1991.

[50] S. S. KIDAMBI, F. EL-GUIBALY et A. ANTONIOU : Area-efficient multipliers for digital signal processing applications. *IEEE Transactions on Circuits and Systems—II : Analog and Digital Signal Processing*, 43(2):90–95, février 1996.

[51] S.-M. KIM, J.-G. CHUNG et K. K. PARHI : Low error fixed-width CSD multiplier with efficient sign extension. *IEEE Transactions on Circuits and Systems—II : Analog and Digital Signal Processing*, 50(12):984–993, décembre 2003.

[52] K.-J. CHO, K.-C. LEE, J.-G. CHUNG et K. K. PARHI : Design of low-error fixed-width modified booth multiplier. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(5):522–531, mai 2004.

[53] T.-B. JUANG et S.-F. HSIAO : Low-error carry-free fixed-width multipliers with low-cost compensation circuits. *IEEE Transactions on Circuits and Systems—II : Analog and Digital Signal Processing*, 52(6):299–303, juin 2005.

[54] L.-D. VAN, S.-S. WANG et W.-S. FENG : Design of the lower error fixed-width multiplier and its application. *IEEE Transactions on Circuits and Systems—II : Analog and Digital Signal Processing*, 47(10):1112–1118, octobre 2000.

[55] L.-D. VAN et C.-C. YANG : Generalized low-error area-efficient fixed-width multipliers. *IEEE Transactions on Circuits and Systems—I : Regular Papers*, 52(8):1608–1619, août 2005.

[56] S. F. OBERMAN et M. J. FLYNN : Minimizing the complexity of SRT tables. *IEEE Transactions on VLSI systems*, 6(1):141–149, mars 1998.

[57] J. E. ROBERTSON : A new class of division methods. *IRE Transactions Electronic Computers*, EC-7:218–222, septembre 1958.

[58] K. D. TOCHER : Techniques of multiplication and division for automatic binary compputers. *Quart. J. Mech. Appl. Math.*, 11-part 3:368–384, 1958.

[59] M. D. ERCEGOVAC et T. LANG : *Digital Arithmetic*. Morgan Kaufmann, 2003.

[60] M. J. FLYNN et S. F. OBERMAN : *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.

[61] S. F. OBERMAN et M. J. FLYNN : Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, août 1997.

[62] S. F. OBERMAN et M. J. FLYNN : Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, février 1997.

[63] A. A. LIDDICOAT et M. J. FLYNN : Parallel square and cube computations. *Dans Proc. 34th Asilomar Conference on Signals, Systems & Computers*, volume 2, pages 1325–1329. IEEE, octobre 2000.

[64] P. IENNE et M. A. VIREDAZ : Bit-serial multipliers and squarers. *IEEE Transactions on Computers*, 43(12):1445–1450, décembre 1994.

[65] S. KRITHIVASAN, M. J. SCHULTE et J. GLOSSNER : A subword-parallel multiplication and sum-of-squares unit. *Dans Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 19–20. IEEE Computer Society, février 2004.

[66] B. PARHAMI : *Computer Arithmetic : Algorithms and Hardware Designs*. Oxford University Press, 2000.

[67] J.A. PIÑEIRO, J. D. BRUGUERA et J.-M. MULLER : Faithful powering computation using table look-up and a fused accumulation tree. *Dans Proc. 15th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 40–47. IEEE Computer Society, juin 2001.

[68] C.-L. WEY et M.-D. SHIEH : Design of a high-speed square generator. *IEEE Transactions on Computers*, 47(9):1021–1026, septembre 1998.

[69] K. E. WIRES, M. J. SCHULTE, L. P. MARQUETTE et P. I. BALZOLA : Combined unsigned and two's complement squarers. *Dans Proc. 33th Asilomar Conference on Signals, Systems & Computers*, volume 2, pages 1215–1219. IEEE, octobre 1999.

[70] B. R. LEE et N. BURGESS : Improved small multiplier based multiplication, squaring and division. *Dans Proc. 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 91–97. IEEE Computer Society, avril 2003.

[71] A. J. AL-KHALILI et A. HU : Design of a 32-bit squarer — exploiting addition redundancy. *Dans Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume V, pages 325–328. IEEE, mai 2003.

[72] N. J. HIGHAM : *Handbook of Writing for the Mathematical Sciences*. SIAM, second édition, 1998.

[73] L. CORDESSES : Direct digital synthesis : A tool for periodic wave generation (part 1). *IEEE Signal Processing Magazine*, 21(4):50–54, juillet 2004.

[74] E. GOUBAULT, M. MARTEL et S. PUTOT : FLUCTUAT : Static analysis for numerical precision. http://www-list.cea.fr/labos/fr/LSL/fluctuat/. CEA-LIST.

[75] J.-M. CHESNEAUX, L.-S. DIDIER, F. JÉZÉQUEL, J.-L. LAMOTTE et F. RICO : CADNA : Control of accuracy and debugging for numerical applications. http://www-anp.lip6.fr/cadna/. LIP6–Univ. Pierre et Marie Curie.

[76] D. MÉNARD et O. SENTIEYS : Automatic evaluation of the accuracy of fixed-point algorithms. *Dans* C. D. KLOOS et J. da FRANCA, éditeurs : *Proc. Design, Automation and Test in Europe (DATE)*, pages 529–537, mars 2002.

[77] N. BRISEBARRE, J.-M. MULLER et A. TISSERAND : Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, 32(2):236–256, juin 2006.

[78] N. BRISEBARRE, J.-M. MULLER et A. TISSERAND : Sparse-coefficient polynomial approximations for hardware implementations. *Dans Proc. 38th Asilomar Conference on Signals, Systems and Computers*, pages 532–535, Pacific Grove, California, U.S.A., novembre 2004.

Annexes

RENPAR'16 / CFSE'4 / SympAAA'2005 / Journes Composants
Le Croisic, France, 5 au 8 avril 2005

Évaluation de polynômes et de fractions rationnelles sur FPGA avec des opérateurs à additions et décalages en grande base

Romain Michard, Arnaud Tisserand et Nicolas Veyrat-Charvillon

Projet Arénaire, LIP, ENS Lyon, 46 allée d'Italie, 69364 LYON Cedex 07
{prénom.nom}@ens-lyon.fr

Résumé
Ce travail porte sur l'étude et l'implantation FPGA d'opérateurs arithmétiques, à base d'additions et de décalages, pour l'approximation polynomiale et rationnelle de fonctions. Ces opérateurs sont des versions en grande base de l'itération de E-méthode proposée par M. Ercegovac dans [4, 6]. Les résultats montrent que ces opérateurs présentent de bonnes performances en alliant la simplicité des architectures à base d'additions et de décalages et le caractère générique des approximations polynomiales et rationnelles.

Mots-clés : arithmétique des ordinateurs, opérateur arithmétique matériel, évaluation de polynôme, évaluation de fraction rationnelle, E-méthode.

1. Introduction

L'implantation matérielle d'opérateurs arithmétiques évolués est un besoin important dans bon nombre d'architectures spécialisées en traitement du signal et des images ou en contrôle numérique. Dans ce papier, nous étudions des opérateurs arithmétiques pour l'évaluation de fonctions algébriques (division, racine carrée...) et de fonctions élémentaires (sinus, cosinus, exponentielle, logarithme, arc-tangente...).
Trois grandes classes d'algorithmes sont utilisées pour l'évaluation des fonctions élémentaires [12] : les algorithmes à base d'approximations polynomiales ou rationnelles, les algorithmes à base de tables et enfin les algorithmes à base d'additions et de décalages.
Les approximations polynomiales ou rationnelles sont essentiellement utilisées en logiciel. Les polynômes sont évalués en utilisant le schéma de Horner. Il est assez simple d'obtenir un polynôme d'approximation d'une fonction $f(x)$ en utilisant l'algorithme de Remes [14]. Des approximations polynomiales et rationnelles des principales fonctions utilisées en calcul scientifique peuvent être trouvées dans [9]. Lorsqu'elles sont utilisées en matériel, on essaye souvent d'utiliser les caractéristiques de coefficients particuliers pour réduire la surface des multiplieurs (voir par exemple [13, 7]). Le principal problème pour l'évaluation des fractions rationnelles est le coût de la division en temps et en surface de circuit. Ces limitations font que seules des solutions à base d'arithmétique sérielle [10] ou de fractions continues [17] sont proposées pour l'approximation rationnelle en matériel.
Les méthodes à base de tables reposent sur l'utilisation de petites tables et d'un petit nombre d'opérations très simples comme des additions. Ces méthodes sont limitées aux petites

précisions, jusqu'à une vingtaine de bits [15, 3]. De plus, ces méthodes sont spécifiques à chaque fonction évaluée. L'implantation d'opérateurs permettant d'évaluer plusieurs fonctions n'est donc pas envisageable avec ces méthodes.
Enfin, il y a la classe des algorithmes à base d'additions et de décalages qui fournissent un chiffre du résultat à chaque cycle de calcul. Dans cette classe, on trouve SRT [5] pour la division et la racine carrée ou CORDIC [16, 18] pour certaines fonctions élémentaires. Ces méthodes permettent de réaliser des opérateurs de taille modérée et présentent une architecture simple. Toutefois, ces méthodes sont spécifiques à un petit nombre de fonctions et ne permettent donc pas d'avoir un opérateur générique. La E-méthode, proposée par M. Ercegovac [4, 6], permet d'évaluer des polynômes et des fractions rationnelles avec une itération à base d'additions et de décalages proche de celle utilisée pour la division et la racine carrée. La E-méthode avait été utilisée pour de l'évaluation de polynômes en arithmétique en-ligne (en série avec les poids forts en tête) dans [8] et en base 2.
Nous présentons dans ce travail une implantation sur circuit FPGA d'opérateurs de E-méthode pour des approximations polynomiales et rationnelles en grande base. Les itérations de la E-méthode ne nécessitant que des additions et des décalages, aucun multiplieur ou diviseur n'est nécessaire dans le circuit final. Le but est ici d'allier le caractère générique des approximations polynomiales ou rationnelles et la simplicité des architectures à base d'additions et de décalages. Les opérateurs obtenus sont facilement réutilisables et fonctionnent en arithmétique parallèle sur des surfaces de circuit modérées.
La section 2 présente quelques rappels sur la E-méthode et les approximations polynomiales et rationnelles. L'étude et l'implantation de la E-méthode sur circuit FPGA et en grande base ($\beta \in \{2, 4, 8\}$) sont présentées dans la section 3. Nous comparons nos implantations avec d'autres travaux à la section 4. Enfin, nous concluons et donnons quelques perspectives en section 5.

2. Rappels sur la E-méthode

La méthode d'évaluation, ou E-méthode, a été proposée par M. Ercegovac dans les années 70 [4, 6]. Cette méthode permet de résoudre certains systèmes linéaires, à diagonale dominante, à l'aide d'une itération simple et régulière à base d'additions et de décalages. Les systèmes linéaires cibles sont de la forme :

$$\begin{pmatrix} 1 & -x & 0 & & \cdots & & & 0 \\ q_1 & 1 & -x & 0 & & \cdots & & 0 \\ q_2 & 0 & 1 & -x & 0 & & \cdots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & & \vdots \\ & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & & & \ddots & \ddots & \ddots & 0 \\ & & & & & \ddots & 1 & -x \\ q_n & 0 & & \cdots & & & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ \vdots \\ \vdots \\ p_{n-1} \\ p_n \end{pmatrix} \quad (1)$$

On note $\mathcal{A}$ la matrice de ce système, b le vecteur second membre et y le vecteur solution. La taille du système est $n + 1$.
Après résolution du système 1, la première composante du vecteur solution y est la valeur au point x de la fraction rationnelle R, de degré n, dont les coefficients du numérateur sont les composantes de b et ceux du dénominateur sont les composantes de la première colonne de $\mathcal{A}$.

C'est à dire que la solution de $\mathcal{A}y = b$ est $y = [y_0, y_1, \ldots, y_n]^t$ telle que

$$y_0 = R(x) = \frac{P(x)}{Q(x)} = \frac{p_n x^n + p_{n-1}x^{n-1} + \cdots + p_0}{q_n x^n + q_{n-1}x^{n-1} + \cdots + 1}.$$

La E-méthode permet donc d'évaluer des fractions rationnelles en un point. En toute généralité, les degrés des polynômes au numérateur et au dénominateur de $R(x)$ peuvent être différents (n est alors le plus grand). En pratique, il semble que d'un point de vue de la précision des approximations réalisées, le cas des degrés égaux (ou très proches) soit à privilégier. En effet, la taille globale du circuit est déterminée par le maximum des degrés. Il est donc intéressant d'utiliser des approximations avec des degrés égaux pour obtenir une meilleure précision pour le même coût matériel.

La formulation du système linéaire 1 pouvant être résolu par la E-méthode impose que le polynôme au dénominateur $Q(x)$ soit tel que $q_0 = 1$. En pratique, cette limitation n'est pas problématique. En effet, il existe des techniques de mise à l'échelle permettant de modifier une fraction rationnelle en une fraction rationnelle dont les coefficients respectent cette contrainte.

On trouve dans [4, 6] les limites sur les valeurs possibles pour les coefficients de la fraction rationnelle et sur l'argument x. Des techniques de mise à l'échelle permettent de limiter l'impact de ces contraintes. En règle générale, l'évaluation d'une fonction élémentaire se fait en deux étapes : la réduction d'argument puis l'évaluation proprement dite [12]. La phase de réduction d'argument permet de se ramener à un petit domaine dans lequel l'approximation utilisée est suffisamment précise.

Dans [2], on trouve une méthode permettant de savoir si une fraction rationnelle est calculable à l'aide de la E-méthode. La bibliothèque MEPLib [1] devrait être capable dans un futur proche de fournir des polynômes et des fractions rationnelles avec des contraintes sur leurs coefficients et répondant aux exigences des algorithmes de réduction d'argument classiques.

En simplifiant légèrement la matrice $\mathcal{A}$, la E-méthode permet aussi d'évaluer des polynômes. En effet, si tous les q_i sont nuls (sauf $q_0 = 1$), alors la première composante du vecteur solution est la valeur au point x du polynôme P, de degré n, dont les coefficients sont les composantes de b. C'est à dire, le polynôme est $Q(x) = 1$ et la solution de $\mathcal{A}y = b$ est alors $y = [y_0, y_1, \ldots, y_n]^t$ avec

$$y_0 = P(x) = p_n x^n + p_{n-1}x^{n-1} + \cdots + p_0.$$

Tous les détails sur le domaine de convergence, les valeurs maximales des coefficients et les relations entre les différents paramètres sont détaillés dans [4, 6, 2].

Avant de présenter l'itération de la E-méthode, nous devons introduire quelques notations utiles pour la suite. La base du système de représentation des nombres est notée β (en pratique, $\beta \in \{2, 4, 8\}$ dans ce travail). Le vecteur des restes partiels, de taille $n + 1$, est noté w. Les différentes valeurs d'une quantité dans le temps sont représentées avec la notation crochet (comme en traitement du signal). Par exemple $w[j]$ dénote le vecteur des restes partiels à la $j$ième itération. Le vecteur de chiffres du résultat trouvé à chaque itération j est noté $d[j]$.

L'algorithme de E-méthode est présenté en figure 1. Le vecteur des restes partiels est initialisé avec les coefficients du polynôme P (les composantes de b). Le premier vecteur de chiffres du résultat est le vecteur nul.

Comme tous les algorithmes à base d'addition et de décalages, la E-méthode produit un chiffre du résultat à chaque itération en commençant par les poids forts. La concaténation des différents chiffres fournit une valeur qui tend vers la valeur mathématique du résultat (à l'infini). Ici, le résultat de chaque itération est un vecteur de chiffres $d[j]$. L'itération est basée sur un calcul similaire à celui d'une division où l'on "diviserait" par la matrice $\mathcal{A}$ (d'où le

1 *initialisation :*
2 $w[0] \longleftarrow b$
3 $d[0] \longleftarrow 0$
4 *itération :*
5 **pour** j **de** 1 à m **faire**
6 $w[j] \longleftarrow \beta \times \Big(w[j-1] - \mathcal{A} \times d[j-1] \Big)$
7 $d[j] \longleftarrow S(w[j])$
8 *résultat :*
9 $y_0[m] = \sum_{i=1}^{m} d_0[i]\beta^{-i}$

FIG. 1 – Algorithme d'évaluation avec la E-méthode (version vectorielle).

terme reste partiel pour w). Pour chaque ligne $i = 1, \ldots, n-1$ de la matrice $\mathcal{A}$, le calcul effectué est :

$$w_i[j] = \beta \times \big(w_i[j-1] - d_0[j-1]q_i - d_i[j-1] + d_{i+1}[j-1]x \big) \tag{2}$$

Dans les cas $i = 0$ et $i = n$, le calcul se simplifie en $w_0[j] = \beta \times \big(w_0[j-1] - d_0[j-1] + d_1[j-1]x \big)$ et $w_n[j] = \beta \times \big(w_n[j-1] - d_0[j-1]q_n - d_n[j-1] \big)$.
Le calcul des nouveaux termes du reste partiel n'implique que des additions/soustractions et des produits d'un nombre par un seul chiffre. On verra en section 3 ce qu'il se passe lorsque β augmente.
A chaque itération, un nouveau vecteur de chiffres du résultat $d[j]$ est produit. Ce calcul se fait en utilisant la fonction de sélection S définie dans un cadre général par l'expression 3. Nous présentons en section 3 les détails de la fonction de sélection pour les différentes bases utilisées dans ce travail. Le paramètre ρ est la plus grande valeur possible pour un chiffre.

$$S(x) = \begin{cases} \text{signe } x \times \lfloor |x| + 1/2 \rfloor, & \text{si } |x| \leq \rho \\ \text{signe } x \times \lfloor |x| \rfloor, & sinon, \end{cases} \tag{3}$$

3. Etude et implantation FPGA de la E-méthode

3.1. Architecture des opérateurs

Le calcul de l'itération présentée dans l'algorithme 1 est découpé suivant les lignes de la matrice $\mathcal{A}$. Le calcul correspondant à chaque ligne de l'itération, équation 2, est confié à une unité fonctionnelle de calcul. L'architecture générale de ces unités est représentée en figure 2.
Le calcul effectué dans chaque unité est assez simple. Il se limite à 2 ou 3 additions/soustractions, des multiplications d'un chiffre par un nombre et la fonction de sélection S. Comme dans la suite nous visons des implantations sur circuits FPGA, nous choisissons de représenter le reste partiel w_i en complément à deux pour bénéficier des lignes d'addition rapide présentes dans les FPGA. Dans l'avenir, nous pensons travailler sur des représentations redondantes des restes partiels pour les implantations ASIC.
Les produits d'un chiffre par un nombre peuvent être particulièrement simples suivant la base utilisée. Dans le cas de la base $\beta = 2$, l'ensemble de chiffres utilisé est $\mathcal{E}_{2,1} = \{-1, 0, 1\}$, le produit se résume alors à un simple multiplexeur et à l'utilisation d'un additionneur/soustracteur. Dans le cas de la base $\beta = 4$, deux ensembles de chiffres sont possibles $\mathcal{E}_{4,2} = \{-2, -1, 0, 1, 2\}$ et $\mathcal{E}_{4,3} = \{-3, -2, -1, 0, 1, 2, 3\}$. Le premier nécessite un multiplexeur plus grand et un décalage

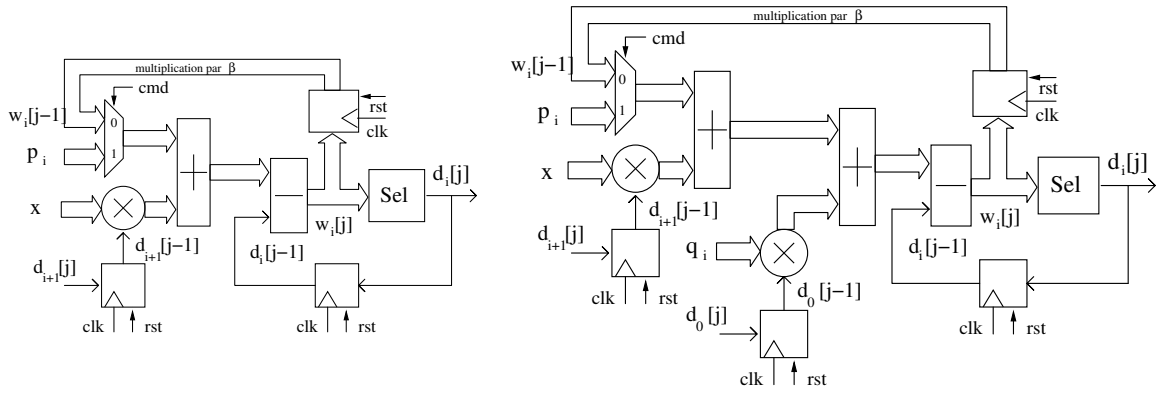


FIG. 2 – Unité fonctionnelle de calcul d'une ligne de l'itération à l'étape j dans le cas polynomial (à gauche) et dans le cas rationnel (à droite).

constant (routage). Dans le cas du second, il faut aussi former le terme $3x$ ou $3q_i$ en utilisant un additionneur ($3x = 2x + x$). On remarque que la formation de ces produits ne se fait qu'une seule fois en début de calcul pour $3x$ et une seule fois à la configuration de Q dans l'unité pour $3q_i$. Le calcul de ces petits produits n'est donc pas un problème.

Le calcul effectué par la fonction de sélection S est représenté dans la table 1 pour les bases 2 et 4 (avec les deux ensembles de chiffres possibles $\mathcal{E}_{4,2}$ et $\mathcal{E}_{4,3}$ pour $\beta = 4$). Le calcul pour la base 8 est similaire mais sa présentation complète nécessite trop de place. La valeur $\hat{w}$ représente une troncation de la valeur w.

On pourrait penser qu'utiliser l'ensemble de chiffres $\mathcal{E}_{4,3}$ en base 4 ne présente pas d'intérêt puisqu'il nécessite une fonction de sélection et des petits produits plus complexes. En pratique ce n'est pas vrai. L'ensemble de chiffres utilisé pour une base donnée influence la largeur de l'intervalle dans lequel se trouve l'argument d'entrée x. Plus l'ensemble de chiffres est vaste plus l'intervalle utilisable pour x est grand (cf. [6, 4]). Un ensemble comme $\mathcal{E}_{4,3}$ en base 4 permet donc d'avoir un domaine d'utilisation de l'approximation plus grand (modulo le fait que le polynôme ou la fraction rationnelle soit choisi en conséquence). En pratique passer de l'ensemble $\mathcal{E}_{4,2}$ à $\mathcal{E}_{4,3}$ permet de doubler la largeur utilisable du domaine de x. Ceci s'explique par la plus grande latitude de correction possible à chaque itération avec les "grands" chiffres supplémentaires.

L'architecture globale de l'opérateur d'évaluation de fraction rationnelle est présentée en figure 3. L'architecture pour la version polynomiale est similaire. Il suffit de remplacer les unités par celles optimisées dans le cas polynomial et de supprimer la mémoire qui stocke les coefficients du polynôme Q.

Dans cet opérateur, le calcul est fait en parallèle sur les $n + 1$ unités pendant m itérations. Les communications pendant le calcul se limitent à la propagation des chiffres $d_i[j]$ d'une unité à la suivante. Au début de chaque nouvelle évaluation, il faut charger la nouvelle valeur de x et réinitialiser les restes partiels avec les coefficients du polynôme P. A chaque changement de fonction à évaluer f, il faut modifier dans les unités les coefficients du polynôme Q.

3.2. Résultats d'implantation des opérateurs

Dans cette section, nous présentons les résultats d'implantation de nos opérateurs en taille et vitesse. Nous avons implanté nos opérateurs sur des FPGA XCV300E Xilinx de la famille Virtex E (avec 3072 slices utilisables au total). Les outils utilisés pour la synthèse et le place-

	$\beta = 2, \mathcal{E}_{2,1}$	
$\hat{w}_i[j-1]$	$d_i[j]$	$w_i[j] \in$
0000	0	$[0, 1/2[$
0001	1	$[1/2, 1[$
0010	1	$[1, 3/2[$
0011	1	$[3/2, 2[$
0100	n/a	impossible
⋮		
1011	n/a	impossible
1100	-1	$[-2, -3/2[$
1101	-1	$[-3/2, -1[$
1110	-1	$[-1, -1/2[$
1111	0	$[-1/2, 0[$

	$\beta = 4, \mathcal{E}_{4,2}$		$\beta = 4, \mathcal{E}_{4,3}$	
$\hat{w}_i[j-1]$	$d_i[j]$	$w_i[j] \in$	$d_i[j]$	$w_i[j] \in$
00000	0	$[0, 1/2[$	0	$[0, 1/2[$
00001	1	$[1/2, 1[$	1	$[1/2, 1[$
00010	1	$[1, 3/2[$	1	$[1, 3/2[$
00011	2	$[3/2, 2[$	2	$[3/2, 2[$
00100	2	$[2, 5/2[$	2	$[2, 5/2[$
00101	2	$[5/2, 3[$	3	$[5/2, 3[$
00110	n/a	impossible	3	$[3, 7/2[$
00111	n/a	impossible	3	$[7/2, 4[$
01000	n/a	impossible	n/a	impossible
⋮				
10111	n/a	impossible	n/a	impossible
11000	n/a	impossible	-3	$[-4, -7/2[$
11001	n/a	impossible	-3	$[-7/2, -3[$
11010	-2	$[-3, -5/2[$	-3	$[-3, -5/2[$
11011	-2	$[-5/2, -2[$	-2	$[-5/2, -2[$
11100	-2	$[-2, -3/2[$	-2	$[-2, -3/2[$
11101	-1	$[-3/2, -1[$	-1	$[-3/2, -1[$
11110	-1	$[-1, -1/2[$	-1	$[-1, -1/2[$
11111	0	$[-1/2, 0[$	0	$[-1/2, 0[$

TAB. 1 – Calcul effectué dans la fonction de sélection S pour la base 2 à gauche et 4 à droite.

ment/routage sont les outils propriétaires Xilinx de l'environnement ISE 5.2i (avec XST pour la synthèse). Les résultats sont obtenus avec un effort d'optimisation normal en vitesse.

Deux fonctions cibles sont utilisées pour ces implantations. La fonction exponentielle $\exp(x)$ sur l'intervalle $[0, 1/8]$ et la fonction sinus $\sin(x)$ sur l'intervalle $[0, \pi/32]$. Différentes précisions cibles sont testées pour chaque fonction et chaque type d'architecture (polynomiale ou rationnelle). A chaque fois, les coefficients sont trouvés par l'algorithme `minimax` de Maple. La précision cible est le nombre de bits corrects pour chaque fonction. En pratique nous utilisons une précision intermédiaire de calcul plus grande pour prendre en compte les erreurs lors de l'évaluation numérique du polynôme. Par exemple, pour l'approximation polynomiale de la fonction exponentielle sur 32 bits, nous utilisons une précision interne de calcul (chemin de données et taille des coefficients) de 37 bits. En règle générale, pour un polynôme de degré d, nous ajoutons d bits de précision interne supplémentaires.

La table 2 présente les différents résultats d'implantation pour la base $\beta = 2$ (donc l'ensemble de chiffres $\mathcal{E}_{2,1}$). Les valeurs reportées dans cette table sont : la précision cible (donnée en nombre de bits corrects), le degré du polynôme ou de la fraction rationnelle utilisé, la taille de l'opérateur obtenu (donnée en nombre de slices du FPGA), et la période de l'opérateur obtenu (donnée en nano-seconde).

La table 3 présente les différents résultats d'implantation pour le cas de la base $\beta = 4$ avec l'ensemble de chiffres entre -2 et 2.

La table 4 présente les différents résultats d'implantation pour le cas de la base $\beta = 4$ avec l'ensemble de chiffres entre -3 et 3.

La table 5 présente les différents résultats d'implantation pour le cas de la base $\beta = 8$ avec l'ensemble de chiffres entre -4 et 4.

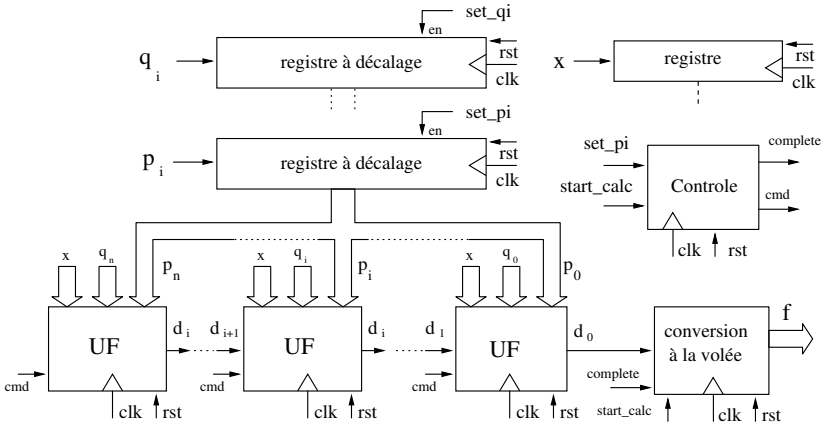


FIG. 3 – Architecture globale de l'opérateur d'évaluation de fraction rationnelle.

Nous illustrons sur les courbes en figure 4 le temps de calcul total pour chacune des solutions polynomiales et chacune des solutions rationnelles. Il est clair sur cette figure que la base 8 permet d'obtenir des temps de calcul totaux bien plus faibles. Cette augmentation de la base se paye en termes de surface.

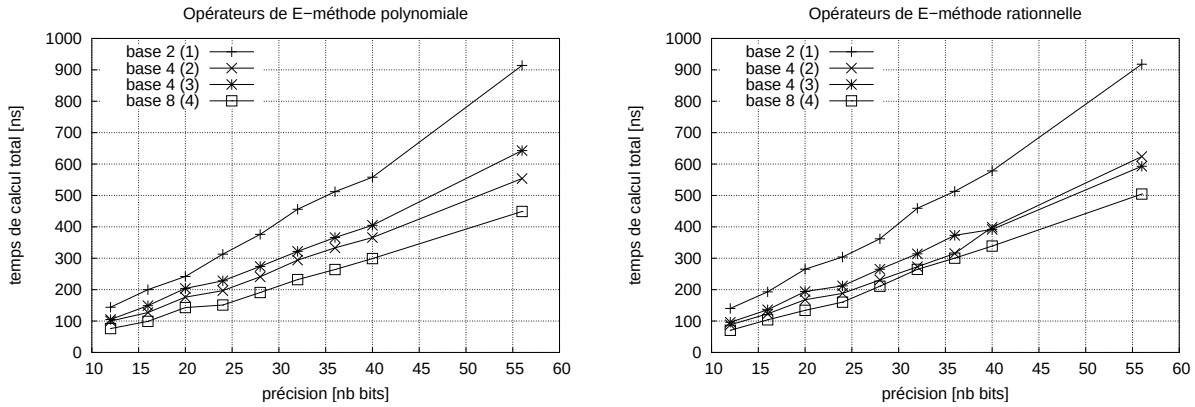


FIG. 4 – Temps de calcul total pour les différentes solutions en fonction de la base (solution polynomiale à gauche et rationnelle à droite).

4. Comparaisons

4.1. Schéma de Horner

Le schéma de Horner permet d'évaluer un polynôme de degré d en utilisant d fois consécutives une brique de base capable de faire un calcul de la forme $XY + Z$ (appelé FMA en arithmétique pour *fused multiply and add*). Dans le cas d'un polynôme de degré 4 on a :

$$P(x) = p_0 + p_1x + p_2x^2 + p_3x^3 + p_4x^4 = p_0 + \Big(p_1 + \big(p_2 + (p_3 + p_4x)x\big)x\Big)x.$$

Fonction $\exp(x)$, base $\beta = 2$, approximation polynomiale									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	3	3	4	4	5	5	6	8
Taille [nb slices]	124	205	253	353	400	529	575	761	1292
Période [ns]	9.92	10.15	9.95	11.78	11.16	12.15	12.53	12.85	14.81

Fonction $\sin(x)$, base $\beta = 2$, approximation polynomiale									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	2	3	3	4	5	5	5	7
Taille [nb slices]	124	153	253	297	400	529	575	624	1135
Période [ns]	10.31	11.10	10.54	11.16	11.75	12.32	12.49	12.41	14.50

Fonctions $\exp(x)$ et $\sin(x)$, base $\beta = 2$, approximations rationnelle									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	1	1	2	2	2	3	3	3	4
Taille [nb slices]	153	186	249	295	331	496	538	590	993
Période [ns]	10.77	11.34	12.05	11.70	12.09	13.11	13.15	13.44	15.30

TAB. 2 – Résultats d'implantation pour la base $\beta = 2$.

Nous avons implanté (dans les mêmes conditions que dans la section 3.2) différents opérateurs d'approximation polynomiale utilisant le schéma de Horner. L'opérateur est composé d'un étage FMA, des registres pour stocker les coefficients du polynôme (les p_i) et la logique de contrôle de l'opérateur. Les résultats d'implantation sont présenté dans la table 6.
On constate que les résultats des opérateurs à base de schéma de Horner sont nettement moins bons que ceux à base de E-méthode (cas polynomial). Par exemple, pour une précision de 32 bits pour la fonction $\exp$, il faut utiliser 1092 slices au lieu de 661 (avec $\beta = 4$ et $\mathcal{E}_{4,3}$) et une période de 30 ns au lieu de 17.5 ns. Soit 65% et 70% d'augmentation respectivement en taille et en période. En comparant avec une version rationnelle des opérateurs de E-méthode en grande base, les écarts sont encore plus grands.
Maintenant ces résultats ne sont pas transposables à tout type de FPGA. Sur les dernières générations de circuit FPGA, il y a des blocs spécialisés pour effectuer des petites multiplications de façon câblée. Pour ces architectures, il serait intéressant de regarder comment se comparent les opérateurs de E-méthode en grande base par rapport à un schéma de Horner utilisant ces petits multiplieurs câblés.

4.2. Schéma de Horner et division SRT

Pour comparer les performances de nos opérateurs dans le cas des fractions rationnelles, nous avons réalisé des opérateurs utilisant le schéma de Horner et un diviseur SRT rapide (en base 4 particulièrement efficace sur FPGA Xilinx). Le diviseur SRT est généré automatiquement à partir d'un outil développé par les auteurs [11]. Les résultats d'implantation sont donnés dans la table 7.
Dans le cas d'une précision de 56 bits, il a été nécessaire de choisir un FPGA plus gros car le nombre de slices disponibles dans un XCV300E n'était pas suffisant (nous avons utilisé un XCV600E pour ce cas particulier).
Du fait du coût important de la division (même si le générateur divgen donne de très bons diviseurs), les résultats pour les approximations rationnelles avec schéma de Horner et division

Fonction $\exp(x)$, base $\beta = 4$ ($\mathcal{E}_{4,2}$), approximation polynomiale									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	3	3	4	4	5	5	6	8
Taille [nb slices]	119	191	238	356	412	568	631	796	2082
Période [ns]	14.35	14.63	15.34	14.96	15.04	15.90	16.25	16.42	17.52

Fonction $\sin(x)$, base $\beta = 4$ ($\mathcal{E}_{4,2}$), approximation polynomiale									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	2	3	3	4	5	5	5	7
Taille [nb slices]	119	149	238	285	412	568	631	665	1834
Période [ns]	14.35	14.17	15.34	14.62	15.04	15.90	16.25	16.25	17.58

Fonctions $\exp(x)$ et $\sin(x)$, base $\beta = 4$ ($\mathcal{E}_{4,2}$), approximations rationnelle									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	1	1	2	2	2	3	3	3	4
Taille [nb slices]	159	203	273	329	378	589	646	719	1574
Période [ns]	13.60	14.47	15.27	14.46	15.42	15.59	16.16	18.58	20.79

TAB. 3 – Résultats d'implantation pour la base $\beta = 4$ avec l'ensemble de chiffres $\mathcal{E}_{4,2} = \{-2, -1, 0, 1, 2\}$.

sont nettement moins bons que ceux avec des opérateurs rationnels de E-Méthode en grande base. Par exemple, pour une précision de 32 bits il faut 1842 slices et une période de 28.4 ns pour le schéma « Horner et division » contre seulement 519 slices et une période de 19.3 ns avec la E-méthode en base 8.

5. Conclusion et perspectives

Ce travail présente l'étude et l'implantation FPGA d'opérateurs arithmétiques, à base d'additions et de décalages, pour l'approximation polynomiale et rationnelle de fonctions en grande base. Ces opérateurs sont des versions en grande base de l'itération de E-méthode proposée par M. Ercegovac dans [4, 6]. Jusqu'ici seules des versions en base 2 de cette méthode étaient proposées. Nous montrons que des bases plus grandes comme 4 ou 8 sont très intéressantes du point de vue de la vitesse tout en se limitant à des tailles modérées de circuit.
Nous pensons travailler, dans le futur, à la réalisation d'un générateur automatique pour ces opérateurs. Nous souhaitons aussi aborder le cas de cibles ASIC, ce qui va nécessiter de prendre en compte des systèmes redondants de représentation des nombres pour toutes les valeurs intermédiaires (les restes partiels w). Ceci risque de fortement compliquer la fonction de sélection, un travail important devra probablement être fait à ce niveau.

Bibliographie

1. Brisebarre (N.), Hennecart (F.), Muller (J.-M.), Tisserand (A.) et Torres. (S.). – MEPLib. – http ://lipforge.ens-lyon.fr/, 2004.
2. Brisebarre (N.) et Muller (J.-M.). – Functions approximable by e-fractions. *In : 38th Conference on signals, systems and computers*. – Pacific Grove, California, US, novembre 2004.

Fonction $\exp(x)$, base $\beta = 4$ ($\mathcal{E}_{4,3}$), approximation polynomiale									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	3	3	4	4	5	5	6	8
Taille [nb slices]	133	220	266	402	462	661	727	934	1677
Période [ns]	15.00	16.94	17.74	17.21	17.15	17.43	17.84	17.77	21.78

Fonction $\sin(x)$, base $\beta = 4$ ($\mathcal{E}_{4,3}$), approximation polynomiale									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	2	3	3	4	5	5	5	7
Taille [nb slices]	133	161	266	316	462	661	727	801	1469
Période [ns]	15.00	16.55	17.74	16.91	17.15	17.43	17.84	18.02	20.40

Fonctions $\exp(x)$ et $\sin(x)$, base $\beta = 4$ ($\mathcal{E}_{4,3}$), approximations rationnelle									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	1	1	2	2	2	3	3	3	4
Taille [nb slices]	183	229	297	353	406	669	736	816	1474
Période [ns]	14.80	16.05	17.61	16.30	17.67	17.94	19.13	18.23	19.78

TAB. 4 – Résultats d'implantation pour la base $\beta = 4$ avec l'ensemble de chiffres $\mathcal{E}_{4,3} = \{-3, -2, -1, 0, 1, 2, 3\}$.

3. de Dinechin (F.) et Tisserand (A.). – Some improvements on multipartite tables methods. *In : 15th International Symposium on Computer Arithmetic ARITH15*, éd. par Burgess (N.) et Ciminiera (L.). pp. 128–135. – Vail, Colorado, juin 2001.
4. Ercegovac (M. D.). – *A general method for evaluation of functions and computation in a digital computer*. – Thèse de PhD, Dept. of Computer Science, University of Illinois, Urbana-Champaign, 1975.
5. Ercegovac (M. D.) et Lang (T.). – *Division and Square-Root Algorithms : Digit-Recurrence Algorithms and Implementations*. – Kluwer Academic, 1994.
6. Ercegovac (M.D.). – A general hardware-oriented method for evaluation of functions and computations in a digital computer. *IEEE Transactions on Computers*, vol. C-26, n° 7, 1977, pp. 667–680.
7. Ercegovac (M.D.), Lang (T.), Muller (J.-M.) et Tisserand (A.). – Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, vol. 49, n° 7, juillet 2000, pp. 627–637.
8. Ercegovac (M.D.), Muller (J.M.) et Tisserand (A.). – FPGA implementation of polynomial evaluation algorithm. *In : Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, éd. par SPIE, pp. 177–188. – octobre 1995.
9. Hart (J.F.). – *Computer Approximations*. – Wiley, 1968.
10. Mencer (O.), Morf (M.), Liddicoat (A.) et Flynn (M. J.). – Efficient digit-serial rational function approximations and digital filtering applications. *In : Asilomar Conference on Signals, Systems, and Computers*, éd. par IEEE. – novembre 1999.
11. Michard (R.), Tisserand (A.) et Veyrat-Charvillon. (N.). – Divgen : a divider circuit generator. – http ://lipforge.ens-lyon.fr/, 2004.
12. Muller (J.-M.). – *Elementary Functions : Algorithms and Implementation*. – Birkhäuser, Boston,

Fonction $\exp(x)$, base $\beta = 8$ ($\mathcal{E}_{8,4}$), approximation polynomiale									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	3	3	4	4	5	5	6	8
Taille [nb slices]	244	408	477	705	801	1075	1386	1583	3070
Période [ns]	16.30	17.89	18.59	17.13	17.86	18.80	19.31	19.85	21.58

Fonction $\exp(x)$, base $\beta = 8$ ($\mathcal{E}_{8,4}$), approximation polynomiale									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	2	3	3	4	5	5	5	7
Taille [nb slices]	244	300	477	551	801	1075	1386	1416	2834
Période [ns]	16.30	16.51	18.59	16.82	17.86	18.80	19.31	19.91	21.38

Fonctions $\exp(x)$ et $\sin(x)$, base $\beta = 8$ ($\mathcal{E}_{8,4}$), approximation rationnelle									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	1	1	2	2	2	3	3	3	4
Taille [nb slices]	58	74	86	100	112	519	573	693	1475
Période [ns]	8.08	9.31	9.81	9.19	9.45	19.30	21.29	20.68	24.16

TAB. 5 – Résultats d'implantation pour la base $\beta = 8$ avec l'ensemble de chiffres $\mathcal{E}_{8,4} = \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$.

1997.

13. Pineiro (J. A.), Bruguera (J. D.) et Muller (J.-M.). – Faithful powering computation using table look-up and a fused accumulation tree. *In : Proceedings of the 15th IEEE Symposium on Computer Arithmetic*. pp. 40–47. – IEEE Computer Society, 2001.
14. Remes (E.). – Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, vol. 198, 1934, pp. 2063–2065.
15. Schulte (M.) et Stine (J.). – Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, vol. 48, n° 8, août 1999, pp. 842–847.
16. Volder (J.). – The CORDIC computing technique. *IRE Transactions on Computers*, vol. EC-8, n° 3, 1959, pp. 330–334.
17. Vuillemin (Jean). – Exact real computer arithmetic with continued fractions. *In : Proc. of the 1988 ACM conference on LISP and functional programming (LFP'88)*. pp. 14–27. – ACM Press, 1988.
18. Walther (J.). – A unified algorithm for elementary functions. *In : Joint Computer Conference Proceedings*. – 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA,1990.

Fonction $\exp(x)$, approximation polynomiale avec schéma de Horner									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	3	3	4	4	5	5	6	8
Taille [nb slices]	175	318	455	642	801	1092	1301	1600	3029
Période [ns]	18.30	21.31	21.15	23.13	24.22	30.06	35.16	29.10	49.34

Fonction $\sin(x)$, approximation polynomiale avec schéma de Horner									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	2	2	3	3	4	5	5	5	7
Taille [nb slices]	175	263	455	591	801	1092	1301	1516	2908
Période [ns]	18.30	22.11	21.15	23.11	24.26	30.06	35.16	29.70	49.09

TAB. 6 – Résultats d'implantation pour le schéma de Horner (approximation polynomiale).

Fonctions $\exp(x)$ et $\sin(x)$, approximation rationnelle avec schéma de Horner et division									
Précision [nb bits]	12	16	20	24	28	32	36	40	56
Degré	1	1	2	2	2	3	3	3	4
Taille [nb slices]	352	524	786	1033	1348	1842	2222	2643	4869
Période [ns]	18.06	24.83	25.02	25.30	26.00	28.34	30.27	36.31	33.41

TAB. 7 – Fonctions $\exp(x)$ et $\sin(x)$, approximation rationnelle avec schéma de Horner et division SRT.

Étude statistique de l'activité de la fonction de sélection dans l'algorithme de E-méthode

Romain Michard, Arnaud Tisserand et Nicolas Veyrat-Charvillon
Projet Arénaire, LIP, ENS Lyon
46 allée d'Italie, 69364 Lyon cedex 07
Mél: {prénom.nom}@ens-lyon.fr

résumé — **Ce travail porte sur l'étude statistique de l'activité liée à la fonction de sélection dans l'algorithme d'approximation de polynômes connu sous le nom de E-méthode proposé par M. Ercegovac dans [1, 2]. La latitude de choix dans la fonction de sélection des chiffres du résultat, en représentation redondante, permet d'envisager de limiter l'activité électrique dans certains cas. Cet article présente un début d'étude sur les gains envisageables dans ce cadre.**

mots clés — **arithmétique des ordinateurs, basse consommation d'énergie, évaluation de polynôme, E-méthode.**

1 Introduction

Dans bon nombre d'applications en traitement du signal et des images, en calcul scientifique ou en contrôle numérique, il est nécessaire d'évaluer des fonctions plus ou moins compliquées en utilisant uniquement des opérateurs simples comme l'addition et la multiplication. Les fonctions algébriques $(1/x, x/y, \sqrt{x}\dots)$ et les fonctions élémentaires ($\sin(x)$, $\cos(x)$, $\exp(x)$, $\log(x)$, $\arctan(x)\dots$) s'approchent assez bien en utilisant des polynômes [3] déterminés, par exemple, avec l'algorithme de Remes [4]. Des approximations polynomiales des principales fonctions utilisées en calcul scientifique peuvent être trouvées dans [5, 3].

La E-méthode, proposée par M. Ercegovac [1, 2], permet d'évaluer des polynômes avec une itération à base d'additions et de décalages proche de celle utilisée pour la division ou dans l'algorithme CORDIC [6, 7]. Dans cet algorithme, les chiffres du résultat sont déterminés de façon itérative (un chiffre à chaque itération) par la fonction de sélection à partir des chiffres précédents et d'un résidu (équivalent au reste partiel dans la division). Les chiffres issus de la fonction de sélection, représentés en notation redondante, offrent une certaine latitude de choix. Dans ce travail, nous étudions l'activité, d'un point de vue statistique, impliquée par la fonction de sélection. Nous proposons une ébauche de méthode permettant de réduire cette activité en utilisant la connaissance du chiffre du résultat sélectionné à la dernière itération.

La section 2 décrit rapidement l'algorithme de E-méthode proposé par Ercegovac. Nous présentons une fonction de sélection avec mémorisation du chiffre de l'itération précédente à la section 3. Dans la section 4, nous présentons les résultats statistiques des simulations fonctionnelles sur l'activité moyenne de la fonction de sélection. Enfin, nous concluons et présentons quelques perspectives à la section 5.

2 Présentation de la E-méthode

La méthode d'évaluation, ou E-méthode, a été proposée par M. Ercegovac dans les années 70 [1, 2]. Cette méthode permet de résoudre certains systèmes linéaires, à diagonale dominante, à l'aide d'une itération simple et régulière à base d'additions et de décalages. Les systèmes linéaires cibles sont de la forme présentée à l'équation (1). Le principal intérêt de cette méthode est de permettre d'évaluer des polynômes sans aucun multiplieur en utilisant un algorithme à base d'additions et de décalages voisin de l'algorithme de division SRT [8, 9]. Ceci permet donc d'envisager des opérateurs de petite taille permettant de limiter la consommation d'énergie statique. Il existe d'autres algorithmes permettant d'évaluer des fonctions algébrique et/ou élémentaires sans multiplieurs comme la méthode des tables multipartites [10]. Mais, en général, ces méthodes sont dédiées à une fonction donnée.

On note $\mathcal{A}$ la matrice de ce système, b le vecteur second membre et y le vecteur solution. La taille du système est $n+1$. Après résolution du système (1), la première composante du vecteur solution y est la valeur au point x du polynôme P, de degré n, dont les coefficients sont les composantes de b. C'est à dire que la solution de $\mathcal{A}y = b$ est $y = [y_0, y_1, \dots, y_n]^t$ telle que

$$y_0 = P(x) = p_n x^n + p_{n-1} x^{n-1} + \cdots + p_0$$

On trouve dans [1, 2] les limites sur les valeurs possibles pour les coefficients de $P(x)$ et sur l'argument x. Des techniques de mise à l'échelle permettent de limiter l'impact de ces contraintes [11].

Avant de présenter l'itération de la E-méthode, nous devons introduire quelques notations utiles pour

$$\begin{pmatrix} 1 & -x & 0 & & \cdots & & & 0 \\ 0 & 1 & -x & 0 & & \cdots & & 0 \\ 0 & 0 & 1 & -x & 0 & & \cdots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & & \vdots \\ & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & \ddots & \ddots & 0 \\ & & & & & & 1 & -x \\ 0 & & & \cdots & & & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ \vdots \\ \vdots \\ p_{n-1} \\ p_n \end{pmatrix} \quad (1)$$

la suite. La base du système de représentation des nombres est notée β (en pratique, $\beta = 4$ dans ce travail, mais les résultats peuvent être généralisés dans d'autres bases). Le vecteur des résidus, de taille $n+1$, est noté w. Les différentes valeurs d'une quantité dans le temps sont représentées avec la notation crochet (comme en traitement du signal). Par exemple $w[j]$ dénote le vecteur des résidus à la $j$ième itération. Le vecteur de chiffres du résultat trouvé à chaque itération j est noté $d[j]$. En base 4, par exemple, ces chiffres sont dans l'ensemble $\{-3, -2, -1, 0, 1, 2, 3\}$.

L'algorithme de E-méthode est présenté en figure 1. Le vecteur des résidus est initialisé avec les coefficients du polynôme P (les composantes de b). Le premier vecteur de chiffres du résultat est le vecteur nul.

1 *initialisation :*
2 $\quad w[0] \longleftarrow b$
3 $\quad d[0] \longleftarrow 0$
4 *itération :*
5 $\quad$ **pour** j **de** 1 à m **faire**
6 $\quad\quad w[j] \longleftarrow \beta \times \Big(w[j-1] - \mathcal{A} \times d[j-1] \Big)$
7 $\quad\quad d[j] \longleftarrow S(w[j])$
8 *résultat :*
9 $\quad\quad y_0[m] = \sum_{i=1}^{m} d_0[i]\beta^{-i}$

FIG. 1 – Algorithme d'évaluation d'un polynôme avec la E-méthode (version vectorielle).

Comme tous les algorithmes à base d'addition et de décalages, la E-méthode produit un chiffre du résultat à chaque itération en commençant par les poids forts. La concaténation des différents chiffres fournit une valeur qui tend vers la valeur mathématique du résultat (à l'infini). Ici, le résultat de chaque itération est un vecteur de chiffres $d[j]$. En pratique, le calcul est décomposé en étages, où le calcul correspondant à une ligne du système linéaire (1) est effectué par un étage. Les chiffres du vecteurs $d[j]$ se propagent donc sur les étages et le résultat final $P(x)$ est la sortie du dernier étage. L'itération est basée sur un calcul similaire à celui d'une division où l'on "diviserait" par la matrice $\mathcal{A}$ (w est analogue à un reste partiel). Pour chaque ligne $i = 0, \ldots, n-1$ de la matrice $\mathcal{A}$, le calcul effectué pour chaque composante est :

$$w_i[j] = \beta \times \big(w_i[j-1] - d_i[j-1] + d_{i+1}[j-1]x \big) \quad (2)$$

Dans le cas $i = n$, le calcul se simplifie en $w_n[j] = \beta \times \big(w_n[j-1] - d_n[j-1] \big)$.

On note dans l'équation (2) que le seul produit $d_{i+1}[j-1]x$ est en fait une multiplication d'un chiffre $d_{i+1}[j-1]$ par un nombre x. En pratique, cette petite "multiplication" est faite en sélectionnant le bon multiple de x à ajouter parmi les différents multiples possibles. Ceci est possible car le multiplicande x est constant pendant toute la durée de l'algorithme.

Le calcul des nouveaux termes du résidu n'implique que des additions/soustractions et des produits d'un nombre par un seul chiffre (et petit, ici il est inférieur à 3). A chaque itération, un nouveau vecteur de chiffres du résultat $d[j]$ est produit. Ce calcul se fait en utilisant la fonction de sélection S définie dans un cadre général par l'expression (3), où ρ est la valeur maximum autorisée pour les chiffres du résultat en notation redondante (ici $\rho = 3$). Des détails peuvent être trouvés dans [1, 2].

$$S(x) = \begin{cases} \text{signe } x \times \lfloor |x| + 1/2 \rfloor, & \text{si } |x| \leq \rho \\ \text{signe } x \times \lfloor |x| \rfloor, & \textit{sinon}, \end{cases} \quad (3)$$

3 Fonction de sélection avec mémoire

Afin de limiter l'activité de l'addition du calcul de $w[j] \longleftarrow \beta \times \Big(w[j-1] - \mathcal{A} \times d[j-1] \Big)$, nous proposons une nouvelle fonction de sélection avec mémoire. Pour les plages de valeurs où la fonction de sélection peut retourner deux valeurs pour le nouveau chiffre du résultat (à l'itération j), on retourne le dernier chiffre utilisé (c.a.d. à l'itération $j-1$) si ce choix est possible (illustration en figure 3).

La simplicité de la fonction de sélection permet de l'implanter sous la forme d'un opérateur moins coûteux qu'une table. La fonction de sélection sans mémoire se sert de la redondance pour n'utiliser que la partie entière $E(w)$ du résidu et le premier bit de sa partie fractionnaire $F(w)$. Dans la sélection avec mémoire, on conserve la redondance pour offrir une latitude sur le choix du chiffre, afin de privilégier les cas

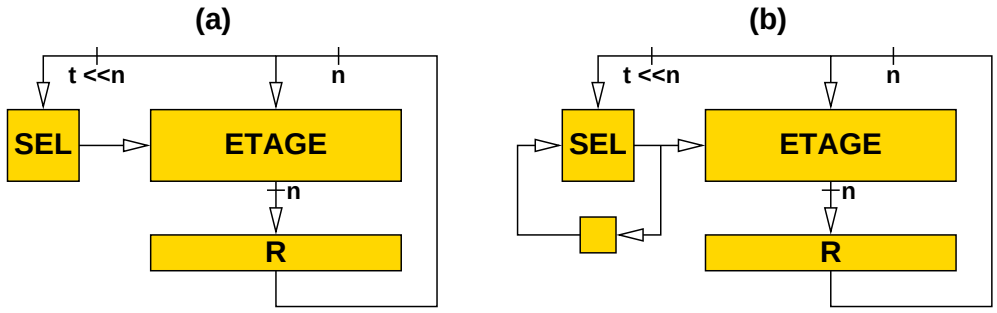


FIG. 2 – Étage de calcul et fonction de sélection sans (a) et avec (b) mémoire.

qui minimisent l'activité ($Sel(w[j]) = Sel(w[j-1])$). Il est alors nécessaire de prendre en compte plus de bits de $F(w)$ (3 pour nos paramètres) lorsque l'on doit borner celui-ci.

Afin de vérifier que la mémorisation ne coûte pas trop cher en pratique, nous avons effectué quelques synthèses sur des circuits FPGA Virtex d'un polynôme de degré 4 avec 32 bits de précision. Les résultats de synthèse sont présentés dans la table 1. En pratique, la mémorisation et la modification de la fonction de sélection pour diminuer l'activité ne coûte que 9% en surface en plus. De plus, en cassant le chemin critique, notre technique permet même de gagner un peu en vitesse (4%). Nous devons maintenant refaire ces tests sur une technologie ASIC.

Solution	Effort Synthèse	Taille [nb. slices]	Période [ns]
Sans Mémoire	surface	423	19.7
	vitesse	750	16.4
Avec Mémoire	surface	461	18.9
	vitesse	812	16.8

TAB. 1 – Résultats de synthèse.

4 Statistiques sur les choix de la fonction de sélection

Dans cette section, nous présentons les résultats de simulation sur les choix effectués par la fonction de sélection dans un cas particulier (la base $\beta = 4$ avec les chiffres dans l'ensemble $\{-3, -2, -1, 0, 1, 2, 3\}$). Les plages de redondance possibles sont illustrées en figure 4. On constate bien les intervalles où deux chiffres sont possibles.

Nous avons écrit un programme réalisant les différents calculs de l'algorithme donné ci-dessus et permettant d'extraire des statistiques sur l'utilisation de la fonction de sélection. Ces statistiques sont faites sur 500 évaluations en différents points avec une précision de 16 bits et un polynôme de degré 4. La table 2 présente ces statistiques. Pour chaque ligne de la table, le chiffre du résultat $d_i[j]$ est sélectionné un certain nombre de fois suivant le chiffre du résultat

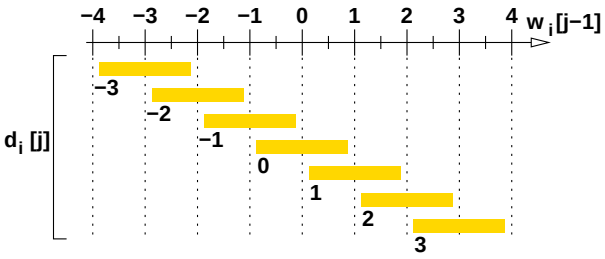


FIG. 4 – Redondance possible pour $\beta = 4$ et les chiffres dans $\{-3, -2, -1, 0, 1, 2, 3\}$.

à l'itération précédente $d_i[j-1]$ en colonne. Les résultats sont normalisés sur le produit du nombre d'itérations (la précision) et du degré du polynôme. La case en (0,0) dépasse la valeur 1 car le chiffre 0 apparaît de très nombreuses fois (par exemple lors de l'initialisation) et en particulier, il peut apparaître sur plusieurs lignes de la matrice pour une itération donnée.

La table 3 présente les statistiques obtenues pour les mêmes paramètres et valeurs qu'à la section précédente en utilisant la fonction de sélection à mémoire. On constate bien que le nombre de cas où le chiffre du résultat choisi est k à l'itération j alors qu'il valait déjà k à l'itération $j-1$ est augmenté (ce sont les cases de la diagonale). En particulier, la fréquence du cas (0,0) est augmentée ce qui est une bonne chose pour limiter l'activité due au calcul et l'activité parasite. On peut donc espérer un gain sur l'activité dans l'étage de calcul du fait que ce chiffre ne change pas et l'argument x est aussi constant pour chaque point d'évaluation.

5 Conclusion et perspectives

Ce travail sur l'activité de la fonction de sélection dans la E-méthode montre qu'il est possible de diminuer la consommation de l'opérateur. Maintenant, il est clair qu'il ne s'agit que d'une étude statistique. En particulier, en pratique le léger surcoût lié aux stockage des chiffres $d_i[j-1]$ (2 à 4 bits suivant la base) et du comparateur va augmenter un peu la surface du circuit.

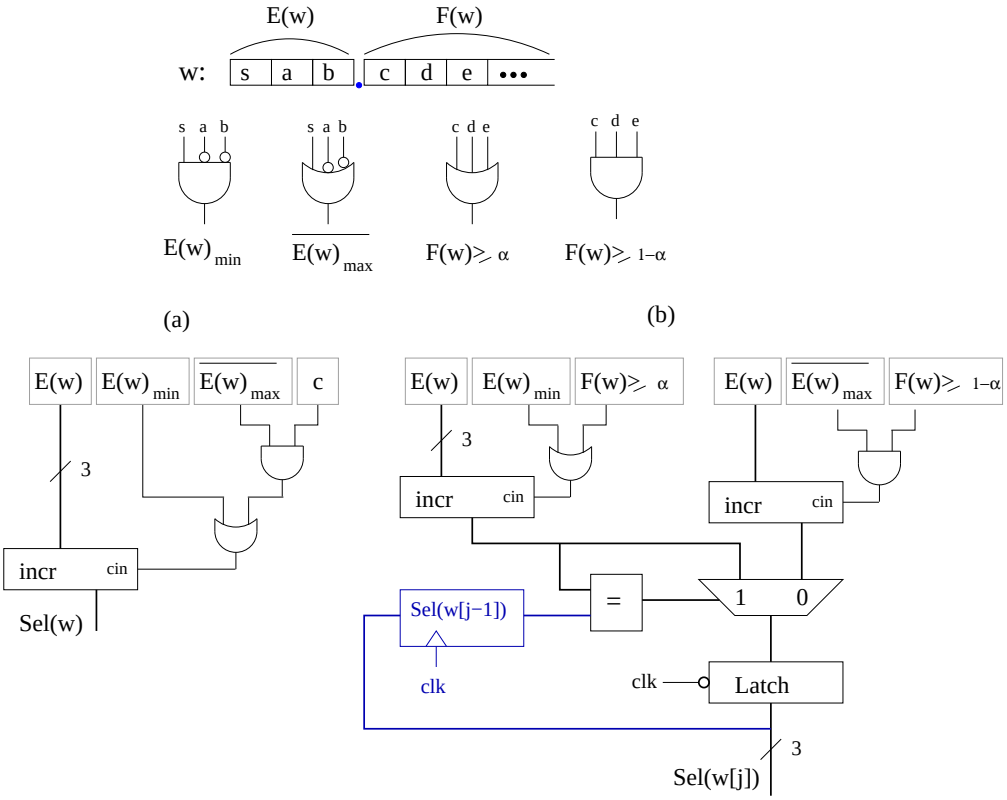


FIG. 3 – Fonction de sélection sans (a) et avec (b) mémoire pour $\beta = 4$ et les chiffres dans $\{-3, -2-, 1, 0, 1, 2, 3\}$.

		$d_i[j-1]$						
		-3	-2	-1	0	1	2	3
$d_i[j]$	-3	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	-2	0.01	0.02	0.01	0.02	0.03	0.03	0.01
	-1	0.00	0.01	0.10	0.34	0.25	0.02	0.00
	0	0.00	0.01	0.02	2.60	0.45	0.01	0.00
	1	0.03	0.02	0.26	0.58	0.25	0.00	0.01
	2	0.00	0.01	0.04	0.04	0.02	0.00	0.00
	3	0.00	0.01	0.00	0.00	0.00	0.01	0.00

TAB. 2 – Statistiques de la sélection standard.

		$d_i[j-1]$						
		-3	-2	-1	0	1	2	3
$d_i[j]$	-3	0.00	0.00	0.00	0.02	0.01	0.00	0.00
	-2	0.00	0.01	0.00	0.01	0.07	0.00	0.00
	-1	0.00	0.00	0.02	0.03	0.02	0.02	0.02
	0	0.02	0.00	0.02	3.00	0.31	0.01	0.07
	1	0.00	0.00	0.00	0.53	0.49	0.01	0.03
	2	0.00	0.00	0.03	0.17	0.03	0.14	0.01
	3	0.00	0.00	0.02	0.08	0.01	0.02	0.01

TAB. 3 – Statistiques de la sélection avec mémoire.

Nous comptons implanter notre solution sur des technologies ASIC pour pouvoir effectuer des simulations au niveau électrique dans un avenir proche. Nous pensons que cet algorithme est intéressant d'un point de vue de la consommation d'énergie car il présente des implantations avec des surfaces bien moindres qu'en utilisant des solutions à base de multiplieurs. Ceci peut donc être un atout pour les technologies avec des courants de fuites importants. Enfin, nous pensons intégrer cet algorithme dans notre générateur de circuits `divgen` [12].

Références

[1] M. D. Ercegovac. *A general method for evaluation of functions and computation in a digital computer*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, 1975.

[2] M.D. Ercegovac. A general hardware-oriented method for evaluation of functions and computations in a digital computer. *IEEE Transactions on Computers*, C-26(7) :667–680, 1977.

[3] J.-M. Muller. *Elementary Functions : Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[4] E. Remes. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, 198 :2063–2065, 1934.

[5] J.F. Hart. *Computer Approximations*. Wiley, 1968.

[6] J. Volder. The CORDIC computing technique. *IRE Transactions on Computers*, EC-8(3) :330–334, 1959.

[7] J. Walther. A unified algorithm for elementary functions. In *Joint Computer Conference Proceedings*, 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA,1990.

[8] M. D. Ercegovac and T. Lang. *Division and Square-Root Algorithms : Digit-Recurrence Algorithms and Implementations*. Kluwer Academic, 1994.

[9] M.D. Ercegovac, J.M. Muller, and A. Tisserand. FPGA implementation of polynomial evaluation algorithm. In SPIE, editor, *Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, volume 2607, pages 177–188, October 1995.

[10] F. de Dinechin and A. Tisserand. Some improvements on multipartite tables methods. In N. Burgess and L. Ciminiera, editors, *15th International Symposium on Computer Arithmetic ARITH15*, pages 128–135, Vail, Colorado, June 2001. IEEE.

[11] N. Brisebarre and J.-M. Muller. Functions approximable by e-fractions. In *38th Conference on signals, systems and computers*, Pacific Grove, California, US, November 2004.

[12] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Divgen : a divider circuit generator. http ://lipforge.ens-lyon.fr/, 2004.

divgen: a divider unit generator

Romain Michard, Arnaud Tisserand and Nicolas Veyrat-Charvillon
Arénaire project (CNRS – ENS Lyon – INRIA – UCBL)
LIP, ENS Lyon. 46 allée d'Italie. F–69364 LYON Cedex 07, FRANCE

ABSTRACT

In this work, we present a tool that generates division hardware units. This generator, called `divgen`, allows a fast and wide space exploration in circuits that involve division operations. The generator produces synthesizable VHDL descriptions of optimized division units for various algorithms and parameters. The results of our generator have been demonstrated on FPGA circuits.

Keywords: Computer arithmetic, division, SRT, circuit generator, FPGA

1. INTRODUCTION

Hardware support for the evaluation of complicated computations has become an important demand in many current applications. Division is frequently used in scientific computing, digital signal processing and computer graphics. A wide variety of algorithms and parameters for the implementation of division is possible. The choice of a division algorithm and the numerous internal parameters is not a straightforward problem.

`divgen` is a fixed-point divider unit generator. Given a set of parameters and options, it automatically generates an optimized VHDL description of the corresponding divider operator for some specific implementation target constraints. `divgen` is an ongoing research project released under the GPL license. More details and new versions can be found in[1] and.[2]

This article is organized as follows. Section 2 presents definitions and notations. Section 3 details the algorithms used, the parameters and options available in the program. Section 4 presents some internal technical details. Performance results on Xilinx FPGAs are given in Section 5.

2. NOTATIONS

The notations proposed in[3] will be used. x stands for the *dividend*, d for the *divisor*, q for the *quotient* and rem for the *final remainder*. The division operation is defined as:

$$x = q\,d + rem.$$

The division algorithms use radix-r quotient representation (a power of 2). In the following, we will present the case of division of integer values, but our generator handles fixed-point values in arbitrary formats.

Each iteration of the digit-recurrence algorithms used for division computes a new radix-r digit of the quotient, noted q_j at iteration (or step) j. The value $w[j]$ denotes the partial remainder (or residual) obtained at step j. The final quotient is an n-bit value. The quotient after j steps is:

$$q[j] = \sum_{i=1}^{j} q_i r^{j-i},$$

Further author information:
Romain Michard: E-mail: `Romain.Michard@ens-lyon.fr`
Arnaud Tisserand: E-mail: `Arnaud.Tisserand@ens-lyon.fr`
Nicolas Veyrat-Charvillon: E-mail: `Nicolas.Veyrat-Charvillon@ens-lyon.fr`

and the final quotient is $q = q[n]$, with

$$n = \left\lceil \frac{\texttt{q\_size}}{\log_2 r} \right\rceil .$$

The supported number systems are: unsigned format, sign-and-magnitude format, 2's complement format, carry-save format.

When using a redundant representation of numbers, a particular number system is specified by the radix r and the largest digit used α, e.g., 4–2 (resp. 4–3) denotes the radix-4 representation using digits from the set $\{-2, -1, 0, 1, 2\}$ (resp. $\{-3, -2, -1, 0, 1, 2, 3\}$). Only symmetrical digit sets are considered in this work, and with $\frac{r}{2} \leq \alpha < r$.

RCA (ripple-carry adder) denotes a sequential carry-propagate adder, CPA (carry-propagate adder) denotes a standard adder and CSA (carry-save adder) a constant time redundant adder. The RCAs are usually found in FPGA implementations where fast carry-lines are present, and CSA are used in ASIC implementations.

When referring to the options, `typewriter font` is used for the option name, *italic font* for a numerical value, sans-serif font for a choice to make between some possible values. For example when setting the dividend number system and quotient digits radix for SRT division:

`dividend_representation` [unsigned | sign-magnitude | 2s_complement]
`q_radix` [*integer*]

3. SUPPORTED ALGORITHMS AND PARAMETERS

3.1. General divider architecture

After execution of the generator, the output file contains a VHDL description of a set of subcomponents and a top level component called "Divider" (presented in Figure 1 and Figure 2, and its port list in Table 1).

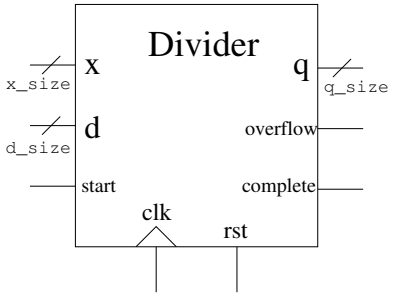


Figure 1. The general architecture of the generated divider

port name	input/output	size	activity	use
x	input	*integer*	N/A	parallel input for dividend
d	input	*integer*	N/A	parallel input for divisor
start	input	1	high	starts the computation
clock	input	1	N/A	clock
reset	input	1	high	resets the operator
q	output	*integer*	N/A	parallel output for quotient
overflow	output	1	high	overflow leading to a false output
complete	output	1	high	computation completion

Table 1. Divider component port list

```
entity Divider is
  port (
    dividend : in std_logic_vector(31 downto 0);
    divisor : in std_logic_vector(15 downto 0);
    clock : in std_logic;
    reset : in std_logic;
    start : in std_logic;
    quotient : out std_logic_vector(15 downto 0);
    overflow : out std_logic;
    complete : out std_logic );
end entity;

architecture arch of Divider is
            [...]
end architecture;
```

Figure 2. Example of the generated top-level entity

Some constraints exist on the input signals: reset should be zero before the start signal is raised. Start should be synchronous and active for exactly one clock period. Dividend and divisor inputs have to be stable from the raise of start until the computation is complete (complete signal raised). When the computation has ended, quotient, overflow and complete stay stable until the next reset signal.

3.2. General options

x, d and q are integers or fixed-point numbers (floating-point numbers are not handled in this version), characterized by their size and their representation (unsigned, sign-and-magnitude and 2's complement). The carry-save format is only possible for the internal residual. The supported algorithms have been tested for sizes between 8 and 64 bits.

Three division algorithms can be selected so far: restoring, nonrestoring and SRT. All are digit-recurrence algorithms, that produce a fixed number of quotient bits at each iteration[3–6] (one quotient digit of one or more bits), most significant digit first. Those algorithms are based on shift and addition operations and some additional logical elements. It is well known that the choice of the radix and quotient digit set influences the overall latency of the algorithm.[3] Roughly, increasing the radix decreases the number of iterations required for the same quotient precision. Unfortunately, as the radix increases, every iteration becomes more complicated and the overall latency may not be reduced as expected. Additionally, it becomes impractical to generate the required divisor multiples for higher radices (costly multiplication in the iteration).

The basic two digit-recurrence algorithms are the *restoring* and the *nonrestoring* algorithms. Those algorithms only rely on very simple radix-2 iterations, i.e., one bit of the quotient is produced at each iteration. The third available algorithm is the SRT division,[7,8] which may rely on a higher radix. This algorithm needs less iterations to compute a quotient but at the cost of a more complicated iteration. Many variations of these algorithms exist, and it is very difficult to decide which one is best suited with respect to some constraints.

The computation of digit-recurrence algorithms is based on a similar recurrence step:

$$\begin{cases} w[0] = x \\ q[0] = 0 \\ q_j = Sel(w[j-1], d) \\ w[j] = rw[j-1] - q_j d \\ q[j] = \sum_{i=1}^{j} q_i r^{j-i} \end{cases}$$

They differ in the radix r, the values that q_j can attain and in the definition of the quotient digit selection function Sel. Another difference between the division algorithms is that the restoring and nonrestoring algorithms

can take any inputs, and will be able to compute, although some inputs may cause an overflow, whereas the SRT division needs a normalized divisor, i.e., we must have:

$$2^{d\_size-1-\delta}\text{ulp}(d) \leq |d| < 2^{d\_size-\delta}\text{ulp}(d) \ ,$$
$$\text{with } \delta = \begin{cases} 0 \text{ if d is } \textit{unsigned} \\ 1 \text{ otherwise.} \end{cases}$$

These are the options common to all algorithms and the values they can take:

- `dividend_representation` [unsigned | sign-magnitude | 2s_complement]
  specifies the number system of the dividend
- `divisor_representation` [unsigned | sign-magnitude | 2s_complement]
  specifies the number system of the divisor
- `quotient_representation` [unsigned | sign-magnitude | 2s_complement]
  specifies the number system of the quotient
- `dividend_size` [*integer*]
  specifies the size of the dividend in number of bits
- `divisor_size` [*integer*]
  specifies the size of the divisor in number of bits
- `quotient_size` [*integer*]
  specifies the size of the quotient in number of bits
- `component_algorithm` [restoring | nonrestoring | SRT]
  the chosen division algorithm

Below, an example of configuration file is presented in case of a 32-bit restoring divider:

```
x_representation unsigned
d_representation unsigned
q_representation unsigned
x_size 32
d_size 32
q_size 32
algorithm restoring
```

3.3. Algorithms and specific options

3.3.1. Restoring division

This algorithm is very similar to the "paper-and-pencil" method learnt in school, using radix $r = 2$ with quotient digit set $\{0, 1\}$. At each iteration, the algorithm subtracts the divisor d from the previous partial remainder $w[j-1]$ multiplied by r . The quotient digit selection function is derived by comparing the residual at each step to 0. If the result of the subtraction is positive (or null), the new digit of the quotient is one, otherwise it is set to zero and the previous value of the partial remainder should be restored (by adding d to the result). Usually, this restoration is not performed using an addition, but by selecting the value $2w[j-1]$ instead of $w[j]$, which replaces an addition in the critical path by a multiplexer. This nonperforming division, presented in Figure 3, is the one generated by `divgen`.

```
1  w[0] ⟵ x
2  for j from 1 to n do
3     if 2w[j − 1] − d ≥ 0 then
4        q_j ⟵ 1
5        w[j] ⟵ 2w[j − 1] − d
6     else
7        q_j ⟵ 0
8        w[j] ⟵ 2w[j − 1]
9  end for
```

Figure 3. Restoring division algorithm

3.3.2. Nonrestoring division

Nonrestoring division[3] is an improved version of the restoring method in the sense that it completely avoids the restoration step by combining restoration additions with the next recurrence, thus, reducing the overall latency. Moreover, it uses the quotient digit set $\{-1, 1\}$ to perform directly the recurrence with the selection function:

$$q_j = Sel(w[j-1]) = \begin{cases} +1 & \text{if } w[j-1] \geq 0, \\ -1 & \text{if } w[j-1] < 0. \end{cases}$$

The nonrestoring division algorithm presented in Figure 4 allows the same small amount of computations at each iteration. The conversion of the quotient from the digit set $\{-1, 1\}$ to the chosen output format can be done either by a subtraction or *on the fly* by using a simple algorithm.[3] We chose that last solution.

```
1  w[0] ⟵ x
2  for j from 1 to n do
3        if w[j − 1] ≥ 0 then
4           w[j] ⟵ 2w[j − 1] − d
5           q_j ⟵ 1
6        else
7           w[j] ⟵ 2w[j − 1] + d
8           q_j ⟵ −1
```

Figure 4. Nonrestoring division algorithm

3.3.3. SRT division

The main problem in division schemes is to determine the new quotient digit q_j at each iteration. The SRT methods use a *redundant quotient digit set* in order to speed up the computation of q_j. More precisely, comparing the partial remainder to all the divisor multiples can offset or possibly diminish the performance gained by increasing the radix. To avoid this, redundancy is introduced in the set of possible quotient digits $|q_i| \leq \alpha$. This allows to simplify the quotient digit selection function in the sense that comparisons are now done with *limited* precision constants only (a constant time operation). This is done using an estimation of the divisor d and the partial remainder $w[j]$.

The hardware implementation is done using a table (or any structure that acts like a table: PLAs, LUTs) addressed by a few of the most significant bits of d and $w[j]$. Some techniques can be used in order to reduce the size and the critical path of the divider, which are described below.

The specific options for the SRT algorithm are:

```
1  if α = r − 1 then
2     w[0] ⟵ x/2
3  else   w[0] ⟵ x/4
4  for j from 1 to n do
5     q_j = Sel(w[j − 1], d)
6     w[j] ⟵ rw[j − 1] − q_j d
7  end for
```

Figure 5. Radix-r SRT division algorithm

- `quotient_radix` [*power_of_2*]
  the value r
- `quotient_max_digit` [*integer*]
  the value α
- `partial_remainder_representation` [2s_complement | carry-save]
  use 2's complement for FPGAs and carry-save for ASICs
- `step_adder` [RCA | CSA]
  use RCA for FPGAs and CSA for ASICs
- `#guard_bits` [*integer*]
  additional bits due to the conversion of the estimation of the redundant residual to a non-redundant form
- `SRT_table_folding` [yes | no]
  folding method proposed in[9] to decrease the selection table area
- `gray_encoding` [yes | no]
  encoding of the selection table output for preserving the redundancy whenever possible

All these options impact the selection function and the whole divider's performances in several ways (see[9] for instance). We will show some examples of situations where they may be useful.

To illustrate the selection function behavior, we use *P-D diagrams*, which are $rw[j]$ versus d diagrams of the valid quotient digit values. The theoretical bounds where the choice of quotient digit $q_j = k$ is valid are the solid plotted lines originating from $(0, 0)$ (they are the U_k and L_k from[3]).

In the following, we will characterize *a priori* the complexity of a selection table by its number of terms (i.e., the number of boxes on the P-D diagram).

- Impact of r and α:
  By increasing the radix r, one decreases the number of iterations. But this also means that more multiples of d have to be computed for the product $q_j d$ and the selection function is more complicated since each iteration must cancel $\log_2 r$ digits of the partial remainder (there are more possible digits for q_j since $\alpha \geq \frac{r}{2}$).

  Increasing α for a constant radix r increases the number of required multiples of d. But this also increases the overlapping of quotient digit selection areas then this leads to a simpler (and smaller) selection function.

  Figure 6 gives a general idea of how we compute the multiples of d. We have $m_0 \in \{-2, -1, 0, 1, 2\}$, $m_1 \in \{-4, 0, 4\}$, $m_2 \in \{-8, 0, 8\}$.

  Figure 7 presents the P-D diagrams for several possible internal representations of the quotient. Table (a) is defined by 272 terms, (b) by 28 and (c) by 400.

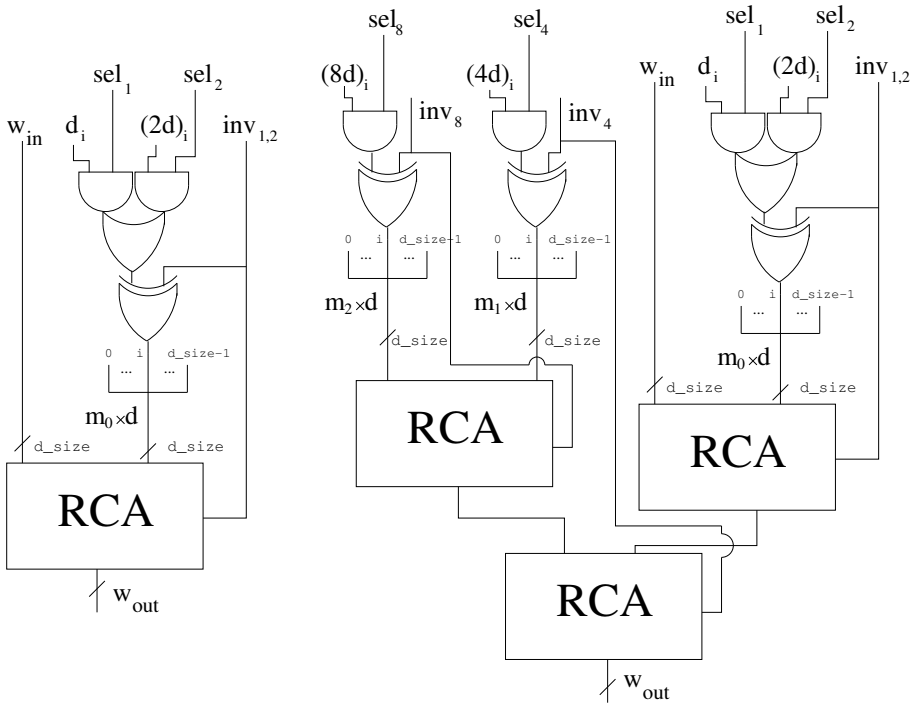


Figure 6. Structures computing $w[j] = rw[j-1] + q_j d$ for $\alpha = 2$ (left) and $10 < \alpha \leq 14$ (right)

- Short CPA based estimation of $w[j]$ and guard bits impact:
  The number of digits used for the estimation $\hat{w}[j]$ of the residual is an important parameter in the selection function. It has to be large enough to ensure a converging algorithm but is should not be too high because the selection table is addressed by $\hat{w}[j]$.

  When using a redundant number system for the residual, the accuracy of the estimation is less than using a non redundant representation. To avoid the use of a selection table with a large number of address bits, one can use a short CPA to get an estimation of the partial remainder $\hat{w}[j]$. It clearly leads to a tradeoff between the additional time required by this short CPA and the time required to decode the address in the selection table. The parameter `#guard_bits` allows to tune the length of the short CPA (see[9] for more details).

  For example, in an SRT 8–6 division, the selection table is defined by 686 terms when using a 2's complement partial remainder. If the partial remainder is represented in carry-save representation, for $g = 0$ we have 2702 terms in the table, 1358 if $g = 1$ and only 686 for $g = 4$.

- Table folding impact:
  Due to the symmetrical digit set being used, the selection table is almost symmetrical with respect to $w[j] = 0$. "Almost" because the truncation error done while estimating the partial remainder is not symmetrical for 2's complement and carry-save number system (it would be for a borrow-save one). We can try to fold the table over the $w[j] = 0$ to try to diminish its size. This is almost always possible, sometimes at the cost of a better estimation made by increasing the number of bits from d and $w[j]$ used to drive the selection table, which can offset and even cancel the gain made by folding. The result is often better than expected, because instead of using $|\hat{w}[j]|$ as the table input, which needs a carry-propagation addition in order to be computed, we use $\hat{w}'[j]$ where $\hat{w}'_i[j] = \hat{w}_i[j] \oplus \text{sign}(\hat{w}[j])$. Doing this for a 2's complement representation, while being faster than taking the actual absolute value of $\hat{w}[j]$, introduces an error that compensates the truncation error made before.

  For example, a selection table for SRT 4–3 (resp. 4–2) with a 2's complement partial remainder contains

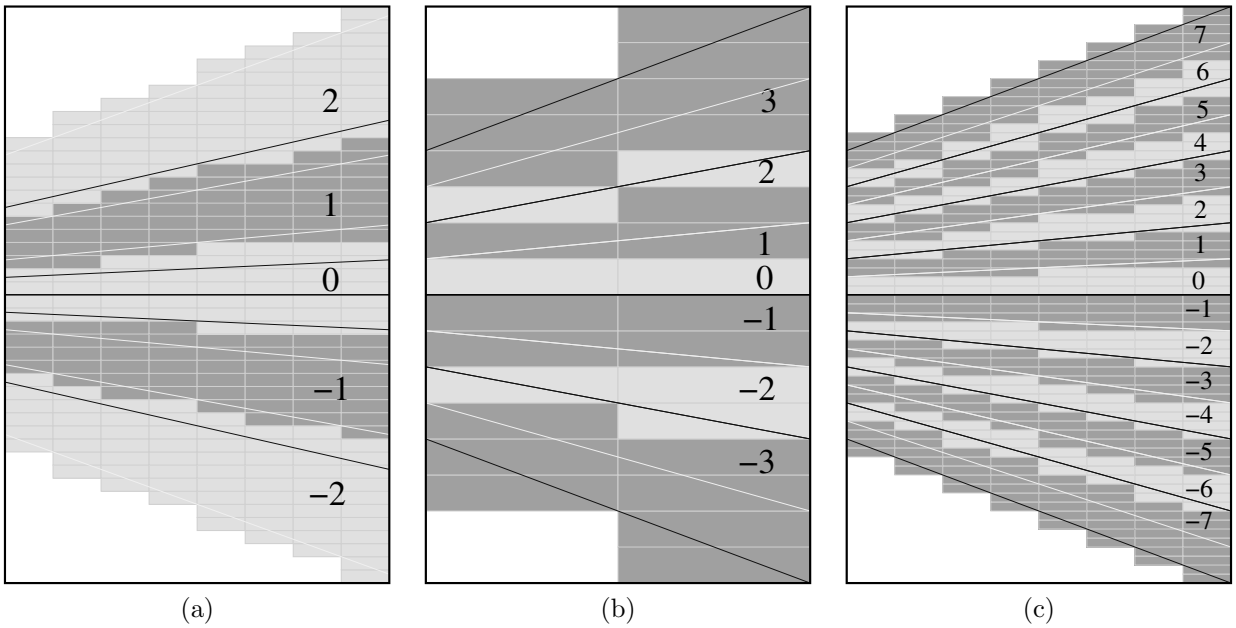


Figure 7. P-D diagrams for several values of r and α: 4–2 (a), 4–3 (b) and 8–7 (c)

28 (resp. 272) terms. If we try to fold it using $|\hat{w}[j]|$ as an input, it needs (a) 52 (resp 269) terms ! Using the XOR-based conversion described above, the table contains only (b) 14 (resp. 136) terms, we halved the table. The corresponding tables are presented in Fig 8.

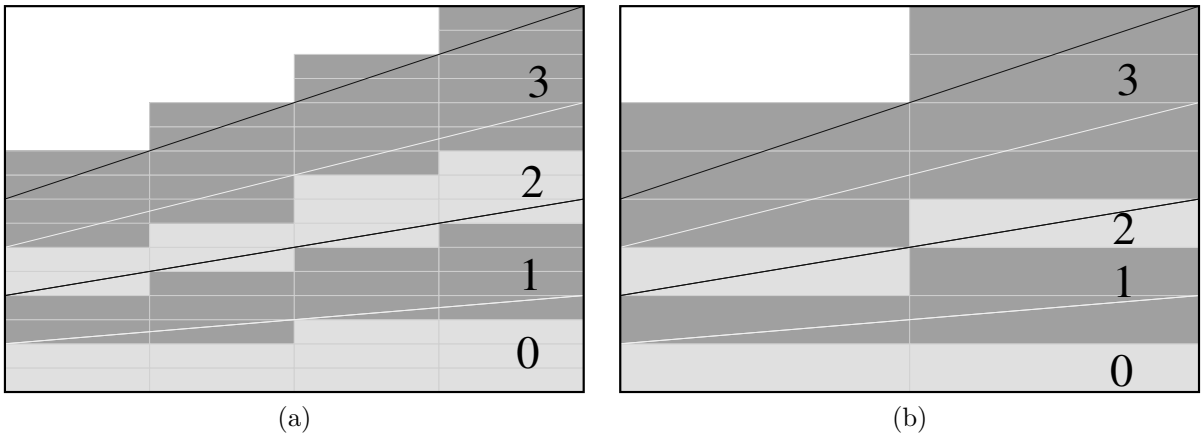


Figure 8. Folded SRT 4–3 tables using different input methods: (a) takes $|\hat{w}[j]|$ and (b) takes $\hat{w}'[j]$ where $\hat{w}'_i[j] = \hat{w}_i[j] \oplus \text{sign}(\hat{w}[j])$.

4. TECHNICAL DETAILS

`divgen` consists in a set of C++ classes (about 5000 lines in version 0.1) used to create a description of an operator which can be converted to VHDL.

All components are derived from the element class which implements the basic component attributes and methods. An instance of element may contain several other elements linked with each other and with the ports

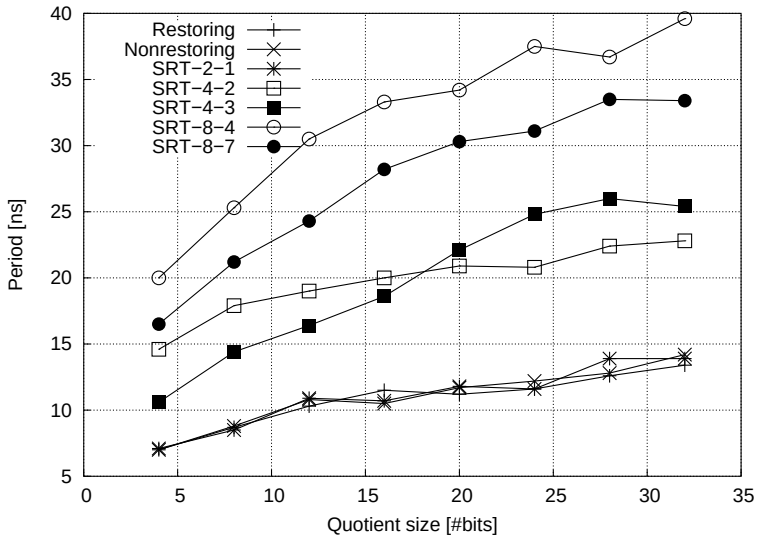


Figure 9. Clock period obtained for several dividers and quotient word lengths

of the surrounding element, possibly included in iterative or conditional structures. A component is instantiated by setting his ports and generic parameters. Wires and sets of wires are related to the signal class, for which some operators have been overloaded in order to allow an easy manipulation, similar to the VHDL one.

As `divgen` is executed, it first reads the configuration file and creates the desired component along with all its subcomponents in memory. Then the main component is parsed and translated to VHDL. All signal assignations and components instantiations are translated automatically to VHDL.

```
element
   |- logic
   |     |- and2
   |     |- or2
   |     |- ...
   |- sequential
   |     |- register
   |- arithmetic
   |     |- RCA
   |     |- divider
   |     |     |- restoring
   |     |     |- nonrestoring
   |     |     |- SRT
```

5. PERFORMANCE RESULTS

Some implementation results of our generator are presented below. The circuit targets are FPGAs from Xilinx (Virtex-E 300, 3072 slices, medium speedgrade).

The overall performances of the generated dividers are presented for several quotient word lengths and the several division algorithms and parameters. Figure 9 reports the clock period results, Figure 10 reports those for the total computation time, Figure 11 reports those for the area and Figure 12 reports the results for area-time product.

In Table 2 and 3 are reported the impact of various synthesis and place and route efforts on the divider performances (in case of a 32-bit divided by 16-bit values for a SRT 4–3 algorithm). Table 4 presents some examples of parameters impact on the performances.

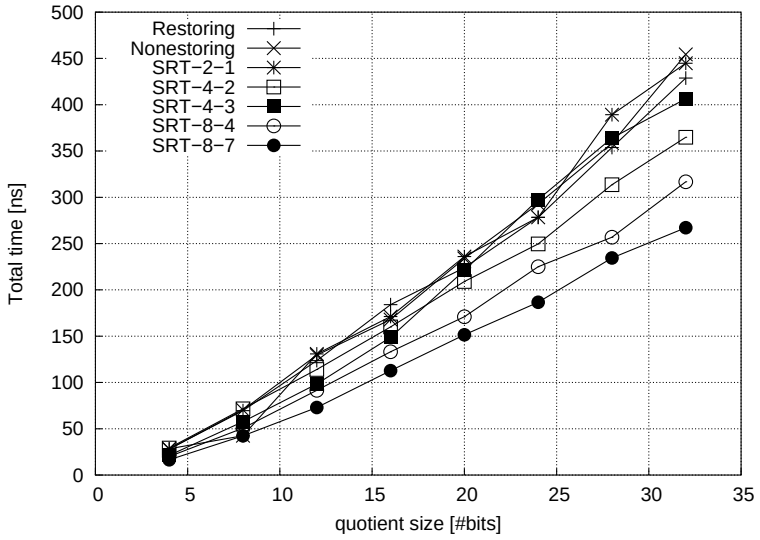


Figure 10. Total computation time obtained for several dividers and quotient word lengths

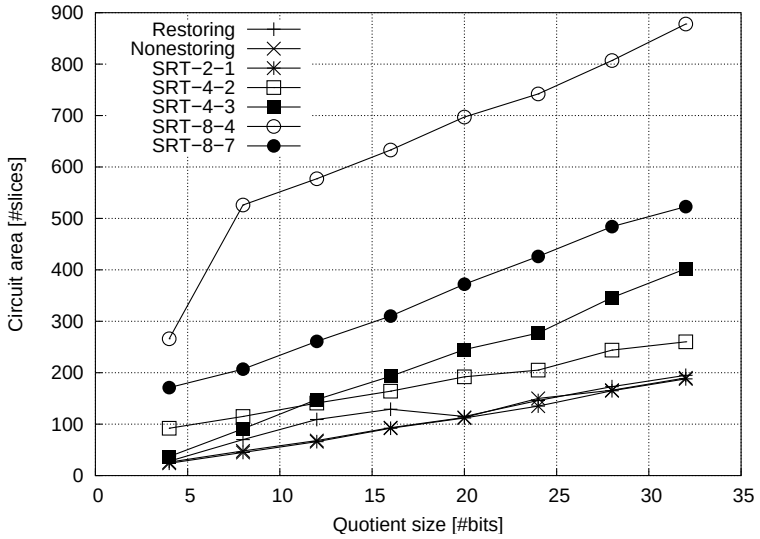


Figure 11. Circuit area obtained for several dividers and quotient word lengths

Synthesis effort	Speed Normal			Speed High		
PnR effort	Standard	Medium	High	Standard	Medium	High
slices	193	193	193	193	193	193
period [ns]	18.6	18.5	18.5	18.6	18.5	18.5

Table 2. Synthesis and Place&Route effort impact for high speed target

Synthesis effort	Area Normal			Area High		
PnR effort	Standard	Medium	High	Standard	Medium	High
slices	100	100	100	100	100	100
period [ns]	26.4	28.2	28.2	26.4	28.2	28.2

Table 3. Synthesis and Place&Route effort impact for small area target

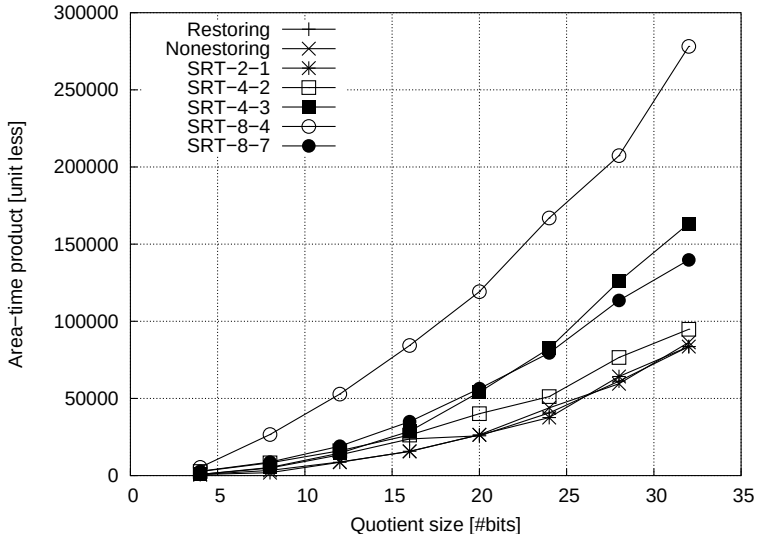


Figure 12. Area-time product obtained for several dividers and quotient word lengths

algorithm	partial remainder	folding	gray encoding	#guard bits	slices	period
2-1	2s	N	N	0	60	23.8
4-2	2s	N	N	0	111	28.1
4-3	2s	N	N	0	100	26.4
8-4	2s	N	N	0		
8-6	2s	N	N	0	220	33.4
8-7	2s	N	N	0	158	34.6
4-3	2s	N	Y	0	104	31.3
4-3	2s	Y	N	0	101	24.3
4-3	2s	Y	Y	0	104	31.6
4-2	2s	N	Y	0	117	28.1
4-2	2s	Y	N	0	95	29
4-2	2s	Y	Y	0	101	27.4
4-3	cs	N	N	0	108	18.7
4-3	cs	Y	N	0	108	20.2
4-3	cs	N	N	1	108	20.1
4-3	cs	N	N	2	108	22.2
4-3	cs	N	N	3	108	23.1
4-3	cs	N	N	4	110	24
4-3	cs	N	N	5	111	21.6
8-4	2s	N	N	0	378	41.4
8-7	2s	Y	N	0	128	31.1
8-7	2s	Y	Y	0	129	31.1

Table 4. Examples of parameters impact

Some numerical tests have been performed in order to validate the behavior of the generated dividers using VHDL simulations. Special cases values and random values have been tested for all the supported algorithms, various parameters, representations and sizes.

6. CONCLUSION & FUTURE PROSPECTS

In this paper, we have presented a tool for the automatic generation of optimized divider hardware units. This program, called `divgen`, generates optimized VHDL descriptions of division operators for various algorithms and parameters with respect to several implementation constraints. The behavior and the performances of the generated dividers have been tested on FPGA circuits from Xilinx.

In the future, we aim at extending this generator to the square root function and other parameters and targets constraints.

REFERENCES

1. R. Michard, A. Tisserand, and N. Veyrat-Charvillon, "Divgen reference manual," research report, Laboratoire de l'Informatique du Parallélisme (LIP), 2005.
2. R. Michard, A. Tisserand, and N. Veyrat-Charvillon, "Divgen web site." http://lipforge.ens-lyon.fr/www/divgen/, 2005.
3. M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, 2003.
4. M. D. Ercegovac and T. Lang, *Division and Square-Root Algorithms: Digit-Recurrence Algorithms and Implementations*, Kluwer Academic, 1994.
5. S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers* **46**, pp. 833–854, Aug. 1997.
6. S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Transactions on Computers* **46**, pp. 154–161, Feb. 1997.
7. J. E. Robertson, "A new class of division methods," *IRE Transactions Electronic Computers* **EC-7**, pp. 218–222, Sept. 1958.
8. K. D. Tocher, "Techniques of multiplication and division for automatic binary compputers," *Quart. J. Mech. Appl. Math.* **11-part 3**, pp. 368–384, 1958.
9. S. F. Oberman and M. J. Flynn, "Minimizing the complexity of SRT tables," *IEEE Transactions on VLSI systems* **6**, pp. 141–149, Mar. 1998.

New Identities and Transformations for Hardware Power Operators

Romain Michard [a], Arnaud Tisserand [b], and Nicolas Veyrat-Charvillon [a]

[a]Arénaire Project, LIP (CNRS–ENSL–INRIA–UCBL), École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon, France;
[b]Arith Group, LIRMM (CNRS–UM2), 161 rue Ada, F-34392 Montpellier, France.

ABSTRACT

In this work we present some improvements on hardware operators dedicated to the computation of power operations with fixed integer exponent ($x^3, x^4, \ldots$) in unsigned radix-2 fixed-point or integer representations. The proposed method reduces the number of partial products using simplifications based on new identities and transformations. These simplifications are performed both at the logical and the arithmetic levels. The proposed method has been implemented in a VHDL generator that produces synthesizable descriptions of optimized operators. The results of our method have been demonstrated on FPGA circuits.

Keywords: Computer arithmetic, integer power operation, square, cube, hardware operator, VLSI, FPGA

1. INTRODUCTION

Square, cube or higher order power operations with fixed integer exponent are frequently used in digital signal processing or graphics applications.[1,2] In digital integrated circuit implementations of such powering operations, one usually replaces standard multipliers by dedicated powering operators. These operators use logical identities in the partial products array (PPA). Due to the symmetries and other simplifications in the PPA, the dedicated operators are significantly smaller and faster than standard multipliers.

In this work we use simplifications both at the logical and the arithmetic levels. The state of the art simplifications are extended and generalized. We also introduce new identities that improve the reduction of the PPA. The application order of these identities is a major concern to minimize the circuit area. In the literature on this subject, this problem is not covered. In this paper, we propose a method that allows a fast reduction of the PPA. Our method works for x^n with n a fixed integer larger than or equal to 3.

This paper is organized as follows. The background and previous works are presented in Section 2. The proposed method is detailed in Section 3. Implementation results and comparisons are reported in Section 4. Section 5 concludes the paper.

2. BACKGROUND

2.1. Notations

Here we deal with x^n. The argument x is a w-bit value in radix-2 unsigned fixed-point or integer format. The bits of x are noted $x_{w-1}, x_{w-2}, \ldots, x_0$. The exponent n is a fixed (i.e., constant in time) integer and $n \geq 3$.

Logical formulas are to be read the following way:

- $\overline{a}$ stands for "not a";

Further author information:
Romain Michard: E-mail: Romain.Michard@ens-lyon.fr
Arnaud Tisserand: E-mail: Arnaud.Tisserand@lirmm.fr
Nicolas Veyrat-Charvillon: E-mail: Nicolas.Veyrat-Charvillon@ens-lyon.fr

8	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*ranks*
				x_4	x_3	x_2	x_1	x_0	
			$\times$	x_4	x_3	x_2	x_1	x_0	
				x_4x_0	x_3x_0	x_2x_0	x_1x_0	x_0x_0	
			x_4x_1	x_3x_1	x_2x_1	x_1x_1	x_0x_1		
		x_4x_2	x_3x_2	x_2x_2	x_1x_2	x_0x_2			
	x_4x_3	x_3x_3	x_2x_3	x_1x_3	x_0x_3				
x_4x_4	x_3x_4	x_2x_4	x_1x_4	x_0x_4					

Figure 1. Partial products array (PPA) for the square of a 5-bit unsigned integer.

- $a \wedge b$, also noted ab, stands for "a and b";
- $a \vee b$ stands for "a or b";
- $a \Rightarrow b$ stands for "a implies b".

The partial products array (PPA) is graphically represented as depicted in Figure 1 (case of the square of a 5-bit unsigned integer). All the partial products (PP) in the column of rank r have a weight equal to 2^r.

We need to perform arithmetic operations on the PPs after transformations. For the "conversion" of logical values to binary values, we use the $[s]$ notation [3, p. 23]. If s is any Boolean statement (true or false), then:

$$[s] = \begin{cases} 1, & \text{if } s \text{ is true,} \\ 0, & \text{if } s \text{ is false.} \end{cases}$$

So the "arithmetic" value of $[x_ix_j \vee x_k]$ in the rank-r column of the PPA is 2^r or 0 depending on the truth of $x_ix_j \vee x_k$. Using this notation it is now clear what the value of $[x_ix_j \vee x_k] + [x_lx_m]$ is (no possible confusion between the arithmetic and the logical operations).

A reduction rule is the arithmetic interpretation of a logical identity. Reduction rules use the $[s]$ notation. For instance the commutativity of the logical `and` leads to the reduction rule $[x_ix_j] + [x_jx_i] \longrightarrow 2 \cdot [x_ix_j]$.

A transformation is a sequence of reduction rules. We show in Section 3.3 that the application order of the reduction rules in transformations is an important parameter. The order of reduction rules inside a transformation is denoted by $\mathrm{T}_{x \to y \to z}$ where rule Rx is applied first, then rule Ry and finally Rz.

2.2. Previous Works

2.2.1. Square

We present here some useful results on dedicated square operators. The presented solutions may be used in higher order powering operators. We have to recall that our method, proposed in Section 3, does not improve square operators.

A folded table-based square method is presented in.[4] The table of squares is repeatedly folded using mathematical properties of the squaring. Implementation results in CMOS technology show that this approach provides a large improvement compared to a conventional ROM implementation.

Symmetries of PPA in squaring operation can also be used in bit-serial architectures.[5]

Implementations of multiplication and squaring on FPGAs with specialized 18×18-bit multiplier blocks are proposed in.[6] Even for this heterogeneous architecture, a dedicated squarer is about half the size of a multiplier.

Dedicated square operators can be designed for signed values. A combined unsigned and two's complement squarer is presented in.[7]

Implementation of squarers for ASIC targets using Wallace-tree and carry-select adder for the PPA addition is presented in.[8]

A specialized squarer is about half the size of a direct multiplier, and is faster. It is also smaller than a booth-encoded multiplier, which requires a non-negligible amount of hardware for the operand recoding.

An example of the symmetries in the PPA for the square is presented in Figure 1 in the case of a 5-bit unsigned integer argument. The standard logical identities usually used in square operators are presented in textbooks on computer arithmetic such as [9, p. 221] and [10, p. 201]. The application of these reduction rules transforms the PPA that will give the same output, but with a lower hardware cost and/or a higher speed.

The first two standard identities rely on the properties of the logical `and` operation:

- Idempotency: $x_i \wedge x_i = x_i$. The corresponding reduction rule removes an `and` gate.
- Commutativity: $x_i \wedge x_j = x_j \wedge x_i$. The addition of two such PPs is replaced by a multiplication by two:

$$[x_i x_j] + [x_j x_i] = 2 \cdot [x_i x_j]$$

  Then, the term $2 \cdot [x_i x_j]$ at rank r is replaced by $[x_i x_j]$ at rank $r + 1$.

Those rules are used as much as possible, since they strictly reduce the hardware area.

A third reduction rule can be used:

$$[x_i x_j] + [x_i] = 2 \cdot [x_i x_j] + [x_i \overline{x_j}]. \tag{1}$$

This last rule trades one `and` gate and a half adder for two `and` gates and an inverter. This rule can be used only once in the transformation of the PPA on the highest column of the PPA, in order to reduce its overall height [9, p. 221]. It can also be applied on several columns in order to diminish the length of the final carry-propagate addition [10, p. 201].

2.2.2. Cube

The cube can be computed using a $n \times n$ multiplication followed by a $2n \times n$ multiplication, but some significant simplifications can be done in the PPA of the cube operators.

The case of an optimized cubing operator is presented in.[11] The PPA reductions rely on an extension of the first two identities used for the square:

- Idempotency of the `and` operation: $x_i \wedge x_i \wedge x_i = x_i$;
- Associativity and Commutativity: $x_i \wedge x_j \wedge x_k = x_i \wedge x_k \wedge x_j = x_j \wedge x_i \wedge x_k = \ldots$

After using the corresponding reduction rules, the PPs in the reduced PPA are divided into two groups: PPs of the form $x_i x_i x_i = x_i$, that occur with a multiplicity of 1, and those of the forms $x_i x_i x_j = x_i x_j$ and $x_i x_j x_k$. These occur respectively 3 and 6 times each. The authors of[11] compute the cube by adding the PPs of the first group with the second group multiplied by three. Their results show that the resulting dedicated cubing operator is faster than a multiplier-based implementation, but has a higher hardware cost.

2.2.3. Other Powering Methods

A modification of a piecewise linear approximation is presented in.[12] It computes a power x^p of an operand x. p is in the form $\pm 2^k$ or $\pm 2^{k_1} \pm 2^{k_2}$, where k, k_1 are integers and k_2 is a positive integer. It can be used for accuracies up to 24 bits but no practical results are reported.

In case of a floating-point argument some polynomial approximations can be used. In,[13] a degree-2 minimax polynomial approximation is used. But in this kind of solution there is a complex trade-off between the accuracy (faithful rounding for instance) and the performances.

3. NEW IDENTITIES AND TRANSFORMATIONS

In Section 3.1 we generalize the previous reduction rules. We introduce some new identities in Section 3.2. Some ideas on the application order of reduction rules inside transformations are presented in Section 3.3. Section 3.4 illustrates the transformations results on several reduced PPA examples.

3.1. Generalization of Previous Reduction Rules

The first reduction rule deals with the idempotency of multiple and identical inputs for an `and` gate:

$$[x_i \wedge x_i \cdots \wedge x_i] = [x_i]. \tag{R0}$$

The second reduction rule relies on the associativity and commutativity of the `and` operation. The output of an `and` gate remains the same when permutations and groupings are applied to its inputs, $x_1 \wedge x_2 \wedge x_3 \wedge x_4 = (x_2 \wedge x_4) \wedge (x_1 \wedge x_3)$ for instance. This leads to a general reduction rule:

$$[x_i x_j \cdots x_l] + [x_{\sigma(i)} x_{\sigma(j)} \cdots x_{\sigma(l)}] = 2 \cdot [x_i x_j \cdots x_l], \tag{R1}$$

where σ is a permutation of the indexes $\{i, j, \ldots, l\}$.

The third reduction Eq. (1) used for square operators can be extended straightforwardly the following way:

$$[P \wedge x_i] + [P] = 2 \cdot [P \wedge x_i] + [P \wedge \overline{x_i}],$$

where P is a logical formula. This allows some hardware reduction, but a more general form can be derived. The idea behind this reduction is to add two PPs, P_1 and P_2, where P_1 is always satisfied when P_2 is (i.e., $P_2 \Rightarrow P_1$). The carry bit of their addition is P_2. The sum bit is true whenever P_1 is satisfied and P_2 is not (the inverse cannot be true). So, we deduce a more general reduction rule:

$$\text{if } P_2 \Rightarrow P_1, \text{ then } [P_1] + [P_2] = 2 \cdot [P_2] + [P_1 \wedge \overline{P_2}] \tag{R2}$$

For instance $[x_1x_2x_3] + [x_1] = 2 \cdot [x_1x_2x_3] + [x_1 \wedge (\overline{x_2} \vee \overline{x_3})]$. This reduction rule alone does not seem to reduce the number of PPs. But it replaces the addition of two PPs in the same column ($[x_1x_2x_3] + [x_1]$ in the previous example) by two PPs in consecutive columns ($2 \cdot [x_1x_2x_3] + [x_1 \wedge (\overline{x_2} \vee \overline{x_3})]$ in the previous example). An example of transformation using this reduction rule for the cube is illustrated in Figure 2.

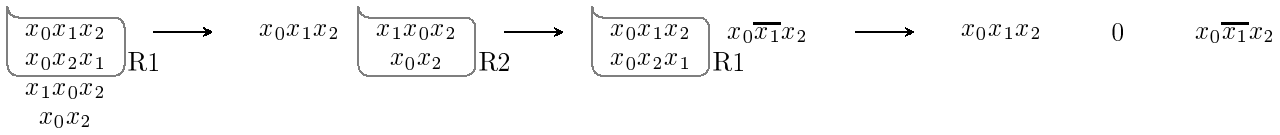


Figure 2. Example of reduction sequence in a cube operator.

3.2. New Identities and Reduction Rules

The transformations based on R2 introduce PPs with complemented bits. In some cases the sum of PPs cannot generate a carry. For instance $[x_1x_2] + [\overline{x_2}x_3] < 2$. Then a half-adder (HA) can be replaced by a 2-input `or` gate. This simplification leads to a new reduction rule for two logical formulas P_1 and P_2:

$$\text{if } \overline{P_1 \wedge P_2}, \text{ then } [P_1] + [P_2] = [P_1 \vee P_2]. \tag{R3}$$

The condition $\overline{P_1 \wedge P_2}$ implies that P_1 and P_2 cannot be true in the same time. Then there is no possible carry and the `xor` gate of the HA can be replaced by a simple `or` gate.

Another reduction rule can be used, it is based on the law of middle excluded:

$$[Q \wedge x_i] + [Q \wedge \overline{x_i}] = [Q], \tag{R4}$$

where Q is a logical formula. Applying this reduction before rule R3, additional reductions are possible.

3.3. Transformations Optimization

As seen in figure 2, the PPs generated using one reduction rule may be used in following reductions. Then, the application order of the reduction rules in transformations is a major concern to minimize the circuit area. This problem seems to not be covered in the literature. It is not computationally feasible to choose the optimal sequence of reductions in the PPA. We study here some possible orders in the transformations.

Figure 3 presents an example of such a difference for the cube of 3-bit values. The upper part of the figure is the initial PPA. The result using the two transformations $T_{0\to1\to2\to4\to3}$ and $T_{0\to2\to4\to3\to1}$ are illustrated on the bottom part of the figure. The result obtained using transformation $T_{0\to1\to2\to4\to3}$ requires an adder from rank 3 up to the most significant column. The transformation $T_{0\to2\to4\to3\to1}$ leads to a solution without addition but more complex logical cells.

Ranks:

8	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

Initial PPA:

		x_2	x_1x_2	x_0x_2	$x_0x_1x_2$	x_0x_2	x_0x_1	x_0
			x_1x_2	x_1x_2	$x_0x_2x_1$	x_0x_1	x_1x_0	
			x_2x_1	x_2x_1	$x_1x_0x_2$	x_2x_0	x_1x_0	
				x_0x_2	x_1	x_0x_1		
				x_2x_1	$x_1x_2x_0$	x_1x_0		
				x_2x_0	$x_2x_0x_1$	x_2x_0		
					$x_2x_1x_0$			

Result using transformation $T_{0\to1\to2\to4\to3}$:

$x_0x_1x_2$	$x_1x_2\overline{x_0}$	$x_2(\overline{x_1}\vee\overline{x_0})$	x_0x_2	x_0x_1	x_1	x_2x_0	x_1x_0	x_0
				$x_2(x_1\vee x_0)$	x_0x_2			

Result using transformation $T_{0\to2\to4\to3\to1}$:

$x_1x_0x_2$	$x_2x_1\overline{x_0}$	x_2	$x_2\overline{x_1}x_0$	$x_2x_0\vee x_1x_0\vee x_1x_2$	$\overline{x_2}x_1\vee\overline{x_0}x_1\vee x_0x_2\overline{x_1}$	x_0x_2	x_0x_1	x_0

Figure 3. Example of initial and reduced PPA for the cube and $w = 3$ using different orders ($T_{0\to1\to2\to4\to3}$ and $T_{0\to2\to4\to3\to1}$).

Figures 4 and 5 illustrates the reduction of column at rank 3 from example Figure 3 using the $T_{0\to1\to2\to4\to3}$ and $T_{0\to2\to4\to3\to1}$ transformations respectively.

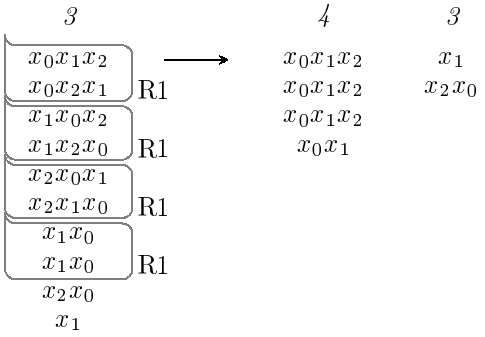


Figure 4. Reduction of rank-3 column in example Figure 3 using transformation $T_{0\to1\to2\to4\to3}$.

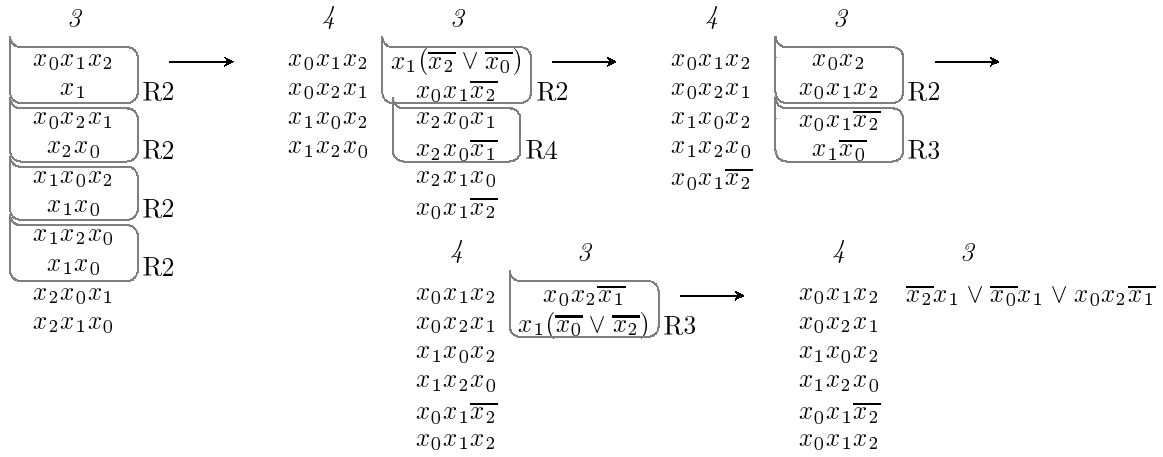


Figure 5. Reduction of rank-3 column in example Figure 3 using transformation $T_{0\to2\to4\to3\to1}$.

Every rule transforms two PPs from rank r into at most one at rank-r and one at rank $r+1$. This means that a good way of applying rules is to process the PPA from the least significant toward the most significant columns. Then, in a column, the number of PPs that can be reduced is maximized.

Each time a couple of PPs is reduced, a new PP may be introduced in the column. The column has to be tested for other reductions. By keeping track of the PPs that have already been tested, the computation time is significantly reduced.

Different orders have been tested. An obvious one is an extension of the textbook method, were our new rules are applied after R1 and R2. This seems a reasonable choice, since the rule R1 is the most hardware efficient, and the new rules R4 and R3 rely on the complemented bits generated by rule R2. This gives transformation $T_{0\to1\to2\to4\to3}$.

The most efficient order found is $T_{0\to2\to4\to3\to1}$. By first applying R2, more complemented bits are introduced in the new PPs, which are then used for R4 and R3. The remaining identical PPs are finally simplified using R1. This order results in less PPs in the reduced PPA. These PPs are usually more complex than those produced by the first order, but this has little impact on LUT-based FPGA implementations.

Table 1 reports the number of PPs obtained using several orders in the case of cube for an operand width of 16 and 24 bits. This shows that the transformation $T_{0\to2\to4\to3\to1}$ seems to be the best one among all the tested transformations.

transformation	$w=16$		$w=24$	
$T_{0\to2\to4\to3\to1}$	997	100%	3723	100%
$T_{0\to4\to3\to1\to2}$	1090	109%	3916	105%
$T_{0\to2\to1\to4\to3}$	1127	113%	4071	109%
$T_{0\to1\to2\to4\to3}$	1134	114%	4062	109%
$T_{0\to4\to3\to2\to1}$	1465	147%	4892	131%

Table 1. Number of PPs using several transformation orders for the cube.

3.4. Reduction Examples

Example of a reduced PPA for the cube and $w=4$:

11	*10*	*9*	*8*	*7*
$x_2x_1x_3$	$x_3(\overline{x_1}x_2 \vee x_0x_1)$	$x_3(x_0\overline{x_1} \vee \overline{x_2x_1} \vee \overline{x_2x_0})$	$x_2(\overline{x_1x_0}x_3 \vee x_1x_0\overline{x_3})$	$x_3(x_1\overline{x_0} \vee x_2\overline{x_1})$
		$x_3(x_1 \vee x_2)$	$\overline{x_0}x_1x_2 \vee x_3x_0\overline{x_1}$	$x_1(x_3\overline{x_0} \vee x_2\overline{x_3})$

6	*5*	*4*	*3*	*2*	*1*	*0*	*(ranks)*
$\overline{x_1}x_3x_0 \vee x_2x_3\overline{x_0} \vee x_2\overline{x_1}$	$\overline{x_1}x_0x_2 \vee x_3x_1\overline{x_0}$	$x_0x_3 \vee x_2x_1\overline{x_0}$ $x_0(x_2 \vee x_1)$	$\overline{x_0}x_1 \vee x_3x_0$ $x_0(\overline{x_1}x_2 \vee x_1\overline{x_2})$	x_0x_2	x_0x_1	x_0	

Example of a reduced PPA for the cube and $w = 5$:

14	*13*	*12*	*11*	*10*
$x_3x_2x_4$	$x_4(\overline{x_2}x_3 \vee x_1x_2)$	$x_4(x_1\overline{x_2} \vee \overline{x_3x_2} \vee \overline{x_3}\overline{x_1})$	$x_4x_0x_3$ $x_3x_4\overline{x_2}$ $x_4(x_3\overline{x_1} \vee x_2\overline{x_1} \vee x_2\overline{x_3})$	$x_1(\overline{x_2x_3}x_4 \vee x_2x_3\overline{x_4})$ $x_4(x_2\overline{x_1} \vee x_3\overline{x_2})$

9	*8*	*7*	*6*
$x_2\overline{x_1}x_4$ $x_3(\overline{x_1}x_2 \vee x_4x_1)$ $x_0(\overline{x_3}x_4 \vee x_1x_3)$ $x_2x_3 \vee x_4x_1\overline{x_2}$ $x_0x_4x_2 \vee x_3\overline{x_4} \vee x_3\overline{x_0}$	$x_2x_0x_3$ $x_4(x_0x_1 \vee x_2\overline{x_0})$ $\overline{x_3}x_4x_0 \vee x_1x_3$	$x_2(x_1\overline{x_4} \vee x_3\overline{x_1})$ $x_2x_0x_1 \vee x_4\overline{x_2x_0}x_1 \vee x_4x_2x_0$ $\overline{x_1}x_3x_0 \vee x_1x_0x_4 \vee x_1x_3\overline{x_0}$	$x_1x_3\overline{x_0}$ $x_4\overline{x_0}x_1$ $x_0x_3\overline{x_1} \vee x_2\overline{x_3} \vee x_2\overline{x_0}$

5	*4*	*3*	*2*	*1*	*0*	*(ranks)*
$x_0\overline{x_1}x_2$ $x_0x_4 \vee x_3x_1\overline{x_0}$	x_0x_3 x_4x_0 $x_1x_2 \vee x_0x_2 \vee x_0x_1$	$\overline{x_0}x_1 \vee x_3x_0$ $x_0(\overline{x_1}x_2 \vee x_1\overline{x_2})$	x_0x_2	x_0x_1	x_0	

Example of a reduced PPA forthe cube and $w = 6$:

17	*16*	*15*	*14*	*13*	*12*
$x_4x_3x_5$	$x_5x_4\overline{x_3}$	$x_5x_2x_3$ $x_5(\overline{x_4} \vee x_1)$	$x_5x_4\overline{x_3}$ $x_5(x_4\overline{x_1} \vee x_3\overline{x_1} \vee x_3\overline{x_4})$	$\overline{x_3}x_2x_5 \vee x_0x_4x_3$ $x_4x_5x_2$ $x_5(x_3\overline{x_2} \vee x_4\overline{x_3})$	$x_4x_1x_2$ $x_4(\overline{x_5} \vee \overline{x_1})$ $x_3(\overline{x_2}x_5 \vee x_4x_2)$ $x_4(\overline{x_0}x_3 \vee x_5x_0)$ $x_0x_3x_5$ $x_5(\overline{x_3}x_2 \vee x_1x_3)$ $x_5(\overline{x_4}x_1 \vee x_2x_4)$

11	*10*	*9*
x_4x_2 $x_5x_1x_2$ $\overline{x_3}x_0x_5 \vee x_4x_3\overline{x_0}$ $\overline{x_4}x_1x_5 \vee x_3\overline{x_1}x_5 \vee x_3x_4x_1$	$\overline{x_2}x_1x_4 \vee x_0x_5x_2$ x_3x_4 $x_4(x_2\overline{x_1} \vee x_3\overline{x_2})$ $\overline{x_3}x_5x_2 \vee x_3\overline{x_5}x_2 \vee x_1x_3x_5$ $x_0\overline{x_4}x_5 \vee x_3\overline{x_0}x_4 \vee x_2x_0x_4$	$x_4x_0\overline{x_2}$ $x_0x_3x_1$ $x_5(\overline{x_1x_0}x_2 \vee x_1x_0\overline{x_2})$ $\overline{x_4}x_3\overline{x_5} \vee \overline{x_4x_0}x_3 \vee \overline{x_1}x_3\overline{x_5} \vee \overline{x_1x_0}x_3 \vee x_1x_4x_0\overline{x_5} \vee x_1x_4x_0\overline{x_3}$ $x_1x_2x_3 \vee x_4\overline{x_1}x_2 \vee x_4x_1\overline{x_2}$

8	*7*	*6*	*5*	*4*
x_5x_1 $x_2(\overline{x_0}x_4 \vee x_5x_0)$ $x_2x_0x_3$ $\overline{x_3x_1}x_4x_0 \vee x_3x_1\overline{x_0} \vee x_3x_1\overline{x_4}$	$x_1x_5\overline{x_0} \vee x_3x_2\overline{x_1}$ $x_4(x_0x_1 \vee \overline{x_2}x_1 \vee x_2x_0)$ $\overline{x_1}x_0x_3 \vee \overline{x_4}x_2x_1$ $x_1(\overline{x_0}x_3 \vee x_2x_0)$	$x_4\overline{x_0}x_1$ $\overline{x_0}x_1x_3 \vee x_5x_0$ $x_0x_3\overline{x_1} \vee x_2\overline{x_3} \vee x_2\overline{x_0}$	x_5x_0 $\overline{x_0}x_3x_1 \vee x_4x_0$ $x_0\overline{x_1}x_2$	x_0x_3 x_4x_0 $x_1x_2 \vee x_0x_2 \vee x_0x_1$

3	*2*	*1*	*0*	*(ranks)*
$\overline{x_0}x_1 \vee x_3x_0$ $x_0(\overline{x_1}x_2 \vee x_1\overline{x_2})$	x_0x_2	x_0x_1	x_0	

4. PRACTICAL RESULTS

Some operators have been generated using the proposed method for x^3 and $w \in \{4, 8, 12, 16, 20, 24\}$ bits. Implementations have been done on Spartan 3 (XC3S1500-4) FPGAs using ISE8.1.03i tools both from Xilinx. Synthesis was area-oriented with a normal effort and place-and-route was processed with a standard effort.

$T_{0\to1}$					
w	#PP	size [slice]	delay [ns]	approx. size	adder cells
4	30	22	9.4	290	22
8	218	153	13.5	2762	202
12	694	527	16.8	9234	670
16	1586	1,223	19.7	21498	1554
20	3022	2,292	21.3	41346	2982
24	5130	3,822	24.5	70570	5082

$T_{0\to1\to2}$					
w	#PP	size [slice]	delay [ns]	approx. size	adder cells
4	26	21	8.5	238	17
8	182	139	15.3	2312	166
12	594	519	16.4	7990	577
16	1390	1,209	18.2	18968	1362
20	2698	2,145	21.1	37140	2666
24	4646	3,799	24.3	64274	4613

Table 2. Implementation results for state of the art methods ([9, p. 221] and [10, p. 201]).

Table 2 reports the implementation results corresponding to the literature methods. The left part is for the rules R0 and R1 only ($T_{0\to1}$). This corresponds to results from the methods provided in textbooks [9, p. 221] or [10, p. 201]. The right part adds rule R2 ($T_{0\to1\to2}$) which is an extension of the state of the art reduction rules.

Table 3 reports the implementation results using our method for two different orders. The left part presents the results for the transformation $T_{0\to1\to2\to4\to3}$ and the right part those for $T_{0\to2\to4\to3\to1}$. Transformation $T_{0\to2\to4\to3\to1}$ seems to be the best one among all the tested transformations even after implementation.

$T_{0\to1\to2\to4\to3}$					
w	#PP	size [slice]	delay [ns]	approx. size	adder cells
4	20	16	6.7	180	10
8	126	120	11.8	1867	110
12	454	445	14.9	6829	436
16	1134	1,128	18.4	16815	1106
20	2294	2,060	21.7	33719	2265
24	4062	3,750	23.2	59267	4030

$T_{0\to2\to4\to3\to1}$					
w	#PP	size [slice]	delay [ns]	approx. size	adder cells
4	17	15	7.1	186	9
8	111	119	14.9	1727	95
12	393	463	15.3	6271	377
16	997	1,068	18.6	15573	971
20	2058	1,991	21.5	31577	2030
24	3723	3,540	23.3	56172	3693

Table 3. Implementation results for our method.

Figure 6 summarizes the implementation results for the cube operators and $w \in \{4, 8, 12, 16, 20, 24\}$ bits. The results are not so good for the larger argument widths. But we think this is due to the FPGA structure, better relative improvement would be obtained on ASIC implementations.

5. CONCLUSION

Some improvements on hardware operators dedicated to the power operation with fixed integer exponent (i.e., $x^3, x^4, \ldots$) have been proposed. The proposed method reduces the number of partial products. It is based on transformations at the logical and the arithmetic levels using new identities and some extension of the state of the art ones. This paper also deals with the application order of the reduction rules in the transformations. The proposed method has been implemented in a VHDL generator that produces synthesizable and optimized operators.

For moderate argument width the proposed method leads to 10–30% area improvement. The area reduction is small for larger argument width. Only a small speed improvement is reported.

The generated operators currently compute the exact value of the result. In the future, we plan to develop truncated version of these operators. We will work on signed representations for this operation. Another interesting future prospect would be the generation of ASIC and low-power consumption versions of these operators. We also plan to apply our results to function approximation methods.[14]

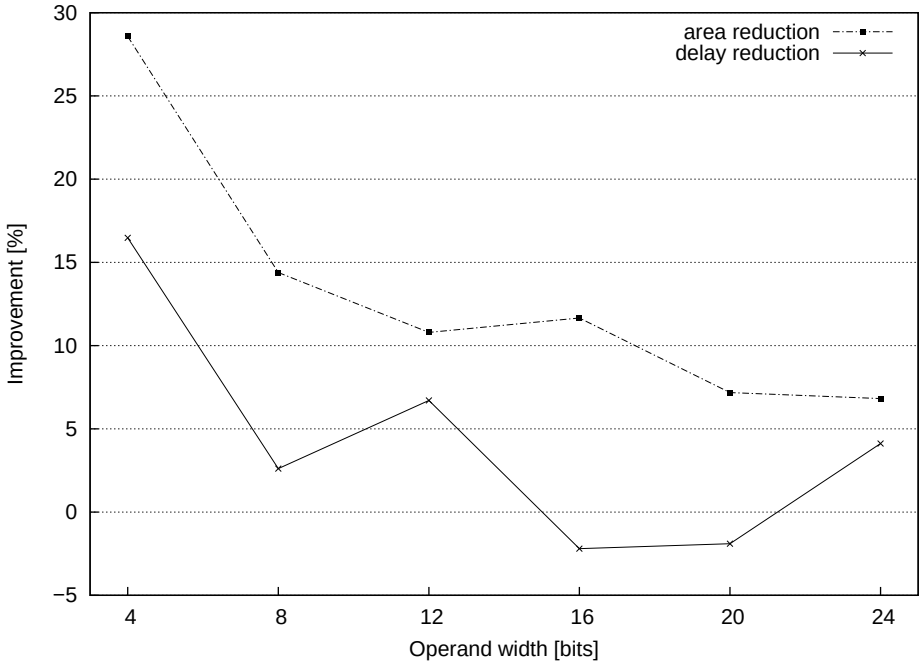


Figure 6. Relative area and speed improvement of our method compared to the state of the art methods.

REFERENCES

1. S. Krithivasan, M. J. Schulte, and J. Glossner, "A subword-parallel multiplication and sum-of-squares unit," in *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 19–20, IEEE Computer Society, Feb. 2004.
2. E. G. Walters and M. J. Schulte, "Efficient function approximation using truncated multipliers and squarers," in *Proc. 17th IEEE Symposium on Computer Arithmetic (ARITH)*, pp. 232–239, IEEE Computer Society, June 2005.
3. N. J. Higham, *Handbook of Writing for the Mathematical Sciences*, SIAM, second ed., 1998.
4. C.-L. Wey and M.-D. Shieh, "Design of a high-speed square generator," *IEEE Transactions on Computers* **47**, pp. 1021–1026, Sept. 1998.
5. P. Ienne and M. A. Viredaz, "Bit-serial multipliers and squarers," *IEEE Transactions on Computers* **43**, pp. 1445–1450, Dec. 1994.
6. B. R. Lee and N. Burgess, "Improved small multiplier based multiplication, squaring and division," in *Proc. 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 91–97, IEEE Computer Society, Apr. 2003.
7. K. E. Wires, M. J. Schulte, L. P. Marquette, and P. I. Balzola, "Combined unsigned and two's complement squarers," in *Proc. 33th Asilomar Conference on Signals, Systems & Computers*, **2**, pp. 1215–1219, IEEE, Oct. 1999.
8. A. J. Al-Khalili and A. Hu, "Design of a 32-bit squarer — exploiting addition redundancy," in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, **V**, pp. 325–328, IEEE, May 2003.
9. M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, 2003.
10. B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.
11. A. A. Liddicoat and M. J. Flynn, "Parallel square and cube computations," in *Proc. 34th Asilomar Conference on Signals, Systems & Computers*, **2**, pp. 1325–1329, IEEE, Oct. 2000.
12. N. Takagi, "Powering by a table look-up and a multiplication with operand modification," *IEEE Transactions on Computers* **47**, pp. 1216–1222, Nov. 1998.

13. J. Piñeiro, J. D. Bruguera, and J. M. Muller, "Faithful powering computation using table look-up and a fused accumulation tree," in *Proc. 15th IEEE Symposium on Computer Arithmetic (ARITH)*, pp. 40–47, IEEE Computer Society, June 2001.
14. R. Michard, A. Tisserand, and N. Veyrat-Charvillon, "Small FPGA polynomial approximations with 3-bit coefficients and low-precision estimations of the powers of x," in *Proc. 16th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 334–339, IEEE Computer Society, July 2005.

Carry Prediction and Selection for Truncated Multiplication

Romain Michard*, Arnaud Tisserand† and Nicolas Veyrat-Charvillon*
* Arénaire project, LIP (CNRS–ENSL–INRIA–UCBL)
Ecole Normale Supérieure de Lyon
46 allée d'Italie. F-69364 Lyon, France
{firstame.lastname}@ens-lyon.fr
† Arith group, LIRMM (CNRS–Univ. Montpellier II)
161 rue Ada. F-34392 Montpellier, France
arnaud.tisserand@lirmm.fr

***Abstract*— This paper presents an error compensation method for truncated multiplication. From two n-bit operands, the operator produces an n-bit product with small error compared to the $2n$-bit exact product. The method is based on a logical computation followed by a simplification process. The filtering parameter used in the simplification process helps to control the trade-off between hardware cost and accuracy. The proposed truncated multiplication scheme has been synthesized on an FPGA platform. It gives a better accuracy over area ratio than previous well-known schemes such as the constant correcting and variable correcting truncation schemes (CCT and VCT).**

I. Introduction

In many digital signal processing applications, fixed-point arithmetic is used. In order to avoid word-size growth, operators with n-bit input(s) must return an n-bit result. For multiplication, the $2n$-bit result of an $(n \times n)$-bit product has to be set back to n-bit by dropping the n least significant bits through a reduction scheme (usually truncation or rounding). This is the purpose of *truncated multipliers*.

Truncated multiplication is used mainly for applications such as finite-impulse response (FIR) filtering and discrete cosine transform (DCT) operations. It can also be used to reduce the hardware cost of function evaluation [1].

This paper starts by the notations and presents the main methods used for truncated multiplication in Section II. In this section, we introduce a simple classification of truncated multiplication schemes. The proposed method is presented in Section III. Our method is based on carry prediction and selection. Section IV presents the error analysis and the implementations results on FPGAs. It also presents a comparison with some existing solutions.

II. Background

A. Notations

Figure 1 presents the partial product array (PPA) of an unsigned (4×4)-bit multiplication (see [2] for full-width multiplication algorithms). The partial product $x_i y_j$ is often represented as a dot for compact notation (see Fig. 2).

We use fixed-point notation with n fractional bits (i.e., $0.x_1x_2x_3 \ldots x_n$) for the operands and the result. As shown in Figure 2, MP represents the $n-1$ most significant columns of the partial product array. MP corresponds to the n bits of the final truncated result. w_{lsb} is the weight of the least significant bit in the truncated result, i.e., $w_{lsb} = 2^{-n}$. The least significant part of the PPA is noted LP, and we further distinguish its k most significant columns as LP_{major}, and the remaining $n-k$ columns as LP_{minor}. In some schemes, the column in LP_{minor} with the highest weight (the left-most column in LP_{minor} in Figure 2) is used. It is referred in the following as $LP^{(h)}_{minor}$. We refer to a column in the PPA by its weight, for example MP extends from columns col_1 to col_n, i.e. from the column where the partial products weight 2^{-1} to the column where the partial products weight $2^{-n} = w_{lsb}$.

			x_3	x_2	x_1	x_0
		$\times$	y_3	y_2	y_1	y_0
			x_3y_0	x_2y_0	x_1y_0	x_0y_0
		x_3y_1	x_2y_1	x_1y_1	x_0y_1	
	x_3y_2	x_2y_2	x_1y_2	x_0y_2		
+	x_3y_3	x_2y_3	x_1y_3	x_0y_3		
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Fig. 1. Partial product array of a (4×4)-bit multiplication.

Function $\text{trunc}_n(x)$ denotes x truncated to the n-th bit, and $\text{round}_n(x)$ stands for x rounded to the n-th bit.

A truncated multiplication scheme computes the partial products in MP, and add an error compensation value (ECV) computed as a function of LP. The result of a truncated multiplication is noted

$$P = \text{trunc}_n\left(MP + f(LP)\right).$$

The most obvious way of performing a truncated multiplication is first to compute the exact $2n$ bits of the result, then round it to n bits. This full-width result is

$$P_{FW} = \text{round}_n\left(MP + LP\right).$$

While giving the smallest possible error, which is only due to the rounding, this method also requires the highest amount

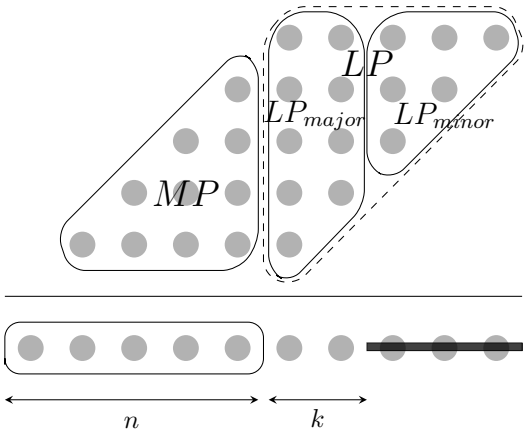


Fig. 2. The different parts of a partial product array.

of hardware by computing all the partial products.

Since the sum bits in the $2n$-bit full-width product are not all used, one is tempted to remove some low-weight columns in LP in order to diminish the hardware cost of the multiplier. However, by doing this, the carries in the low-weight part of the PPA are lost, thereby introducing an evaluation error.

Two kinds of error occur in truncated multiplication: the evaluation error E_{eval}, which is due to the columns that are removed in LP, and the truncation error E_{trunc}, which occurs when the computed value of the PPA is reduced to an n-bit value.

A direct-truncated multiplier computes only the $n-1$ most significant columns of the PPA. While minimizing the required amount of hardware, this approach does not take into account any of the carries propagating from LP, and leads to a maximal evaluation error. The result of a direct-truncated multiplier is

$$P_{DT} = \text{trunc}_n\left(MP\right).$$

B. Classification of the Truncated Multiplication Schemes

The truncated multiplication problem is the trade-off between accuracy and hardware cost. At one side, there is the full-width multiplier with the best accuracy but the highest cost. At the other side, there is the direct-truncated multiplier with the lowest cost but the worst accuracy. Many truncated multiplication schemes have been proposed with intermediate trade-offs.

We propose a simple classification based on the complexity of the ECV computation scheme. We distinguish two main kinds of solutions: *static* ECV and *dynamic* ECV. Static ECV means that the correction value does not depend on the actual values of the operands (the value is fixed at design time). Dynamic ECV uses a correction value computed using the actual operands (at run-time). Obviously, dynamic ECV is more accurate but it requires larger circuit area.

In order to refine the classification, we add subgroups depending on the PPA part impacted by the truncated multiplication method. We propose the five groups defined below.

- Static ECV with C: no partial product from LP is computed, the constant C is based on the LP part.
- Static ECV with $LP_{major}+C$: all partial products from LP_{major} are computed, the constant C is based on the LP_{minor} part.
- Dynamic ECV with $f(LP_{major})$: all partial products from LP_{major} are computed and used to evaluate the correction due to LP.
- Dynamic ECV with $LP_{major}+f(LP_{minor})$: all partial products from LP_{major} are computed, some partial products from LP_{minor} (usually $LP^{(h)}_{minor}$) are used to evaluate the ECV.
- Dynamic ECV with LP: all partial products in LP are computed and used to compute the ECV.

Table I presents the classification of the previous methods and our method accordingly to those groups.

C. Static ECV: C

In [6], the expected value of $C = Sum(LP)$ is estimated by assuming that each bit of the inputs has a probability $1/2$ of being one. The probabilities of each carry and sum bit are evaluated using the logic properties of the half-adder and full-adder cells.

The direct-truncated multiplication scheme described previously also fits in this category with $C = 0$, LP is not computed nor approximated.

D. Static ECV: $LP_{major}+C$

Truncated multiplication schemes with a static ECV approximate the error done by leaving out the low-weight columns LP_{minor} with a constant, which is computed either by exhaustive search on the input values or as a statistical evaluation of the expected value of LP_{minor}. In order to improve the accuracy, the k columns of LP_{major} are used as an extension of MP. The resulting $(n+k)$-bit value is then rounded or truncated to n bits.

In [3], every input bit is assumed to have a probability $\frac{1}{2}$ of being one. Each partial product therefore has an expected value $\frac{1}{4}$. By adding their expected values over LP_{minor}, Lim gets the expected value of the evaluation error:

$$E_{eval} = -\frac{w_{lsb}}{4}\sum_{i=0}^{n-k-1}(i+1)\,2^{i-n}. \quad (1)$$

The multiplication result is:

$$P = \text{round}_n\left(MP + LP_{major} + \text{round}_{n+k}\left(-E_{eval}\right)\right),$$

and the parameter k is chosen so as to give to the evaluation error a variance lower than the variance ${w^2}_{lsb}/12$ of the rounding error, which is treated as a random noise.

This method is refined in the constant correction truncated (CCT) multiplication [7], where the error made by truncating the $(n+k)$-bit result to an n-bit value is computed assuming for each result bit of the multiplication a probability $\frac{1}{2}$ of being

Static ECV		Dynamic ECV		
C	$LP_{major} + C$	$f(LP_{major})$	$LP_{major} + f(LP_{minor})$	LP
direct-truncated [6]	[3] [7]	[4] [8] [11]	[5] [9], [10] our method	full-width

TABLE I
CLASSIFICATION OF THE ECV METHODS USED IN LITERATURE.

one. This gives:

$$E_{trunc} = -\frac{w_{lsb}}{2} \sum_{i=-k}^{-1} 2^i = -\frac{w_{lsb}}{2}(1 - 2^{-k}).$$

The multiplication result is:

$$P = \text{trunc}_n \left(MP + LP_{major} + \text{round}_{n+k} \left(-E_{eval} - E_{trunc} \right) \right).$$

E. Dynamic ECV: $f(LP_{major})$

In order to further diminish the error, some schemes have been proposed where, instead of approximating the value of the partial products in LP_{minor} by a constant, it is expressed as a function of the partial products in LP_{major}.

[4] gives an ECV for a modified Booth encoded PPA, where each line of partial products in LP_{minor} is estimated as a multiple of the corresponding partial product in LP_{major} ($k = 1$). This results in a data-dependent ECV.

[8] presents another dynamic ECV for a modified Booth multiplier. For every possible combination of bits in the recoded operand, a corresponding expected value of LP_{minor} is computed by statistic analysis, and added to the exact value of LP_{major} ($k = 1$). This gives for every possible value of the recoded operand an approximation of the carries propagated from LP. A carry generation circuit is then computed using a Karnaugh map. For sizes larger than 12, the exhaustive simulation is replaced by statistical analysis.

F. Dynamic ECV: $LP_{major} + f(LP_{minor})$

In [5], the ECV for a Baugh-Wooley array multiplier is computed in three parts. First LP_{major} is computed and summed. Then the partial products in $LP^{(h)}_{minor}$ are computed, some of them are inverted, and all this is summed. The pattern of inversions applied to the partial products in $LP^{(h)}_{minor}$ is parametrized by an integer Q. The sum is noted $\theta_{Q,k}$. Finally the expected value of $(LP_{minor} - \theta_{Q,k})$ is estimated, and added do $LP_{major} + \theta_{Q,k}$. The best value of Q is obtained by exhaustive search. For $n \geq 16$, a statistical analysis can be performed.

The variable correction truncated (VCT) multiplication [9], [10] estimates the carries propagated from LP_{minor} by adding to the least column of LP_{major} the partial products of $LP^{(h)}_{minor}$. This is equivalent to multiplying these partial-products by two. An immediate consequence is that the ECV is minimal when the multiplication operands are minimal, and vice versa. The truncation error E_{trunc} is the same as the one defined in the CCT multiplication.

The multiplication result is:

$$P = \text{trunc}_n \left(MP + LP_{major} + 2LP^{(h)}_{minor} + \text{round}_{n+k} \left(-E_{trunc} \right) \right)$$

A hybrid correction truncation (HCT) multiplication [11] realizes a compromise between the CCT and VCT multiplications by only using a percentage p of the partial products in $LP^{(h)}_{minor}$ for the ECV, and adding $1 - p$ of the evaluation error E_{eval}, defined in the CCT multiplication. The truncation error E_{trunc} is also the same as in the CCT and VCT multiplications.

The multiplication result is:

$$P = \text{trunc}_n \left(MP + LP_{major} + p \cdot 2LP^{(h)}_{minor} + \text{round}_n \left((p-1) \cdot E_{eval} - E_{trunc} \right) \right)$$

G. Dynamic ECV: $LP_{major} + LP_{minor}$

Only the full-width multiplier fits into this category: this is the case where all of LP is computed.

III. PROPOSED METHOD

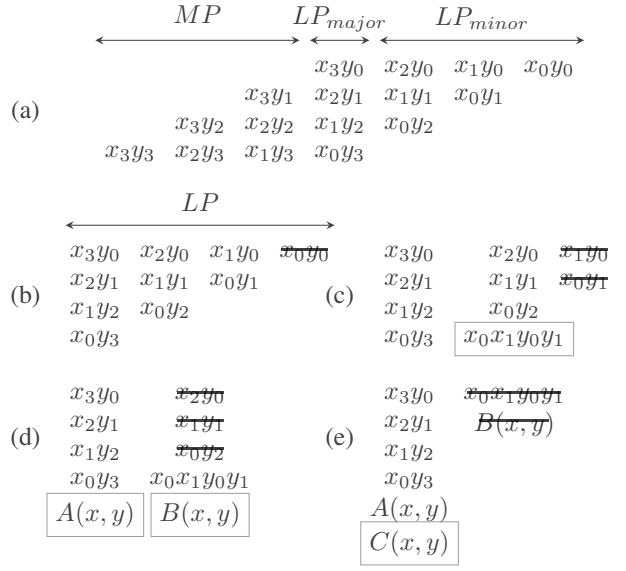
In this work, a new data-dependent truncated multiplication scheme is introduced. It is named *prediction-selection* correcting truncated (PSCT) multiplication. It is proposed for direct non-recoded unsigned array multiplication.

In the CCT, VCT and HCT multiplication schemes, the carries propagated from LP_{minor} are estimated, either by statistical analysis or with the help of the partial products in column $LP^{(h)}_{minor}$. It is then difficult to know what kind of error is done, and what additional terms might be introduced in order to improve accuracy.

Our approach tries to address this issue by computing in a first time the exact values of every carry generated in LP_{minor}, and then discarding the less probable ones. This scheme simplifies the computation of the ECV and lower the associated hardware cost, while keeping track of the error made by removing those products.

A. Carry Prediction

Consider a complete PPA as the one used for the full-width multiplication in Figure 2. Since the $n - k$ least significant bits of the result are discarded, the corresponding sum bits of LP_{minor} do not have to be computed. But if LP_{minor} is



$$A(x,y) = x_0x_1y_1y_2 \vee x_0x_2y_0y_2 \vee x_1x_2y_0y_1$$
$$B(x,y) = (x_0y_2) \oplus (x_1y_1) \oplus (x_2y_0)$$
$$C(x,y) = x_0x_1y_0y_1 \wedge B(x,y) = x_0x_1y_0y_1 \wedge \overline{(x_2 \oplus y_2)}$$

Fig. 3. The four steps of carry prediction for $n = 4$ and $k = 1$.

not implemented at all, all the carries generated there are lost, leading to the evaluation error described in Eq. (1).

In order to keep the evaluation error low while removing unnecessary hardware, only the logic formulas of the carries generated in LP_{minor} have to actually be implemented. These expressions are obtained by replacing the full-adder and half-adder cells in LP_{minor} by their respective logic definitions:

$$\text{carry}_{FA}(a,b,c) = ab \vee ac \vee bc$$
$$\text{sum}_{FA}(a,b,c) = a \oplus b \oplus c$$
$$\text{carry}_{HA}(a,b) = ab$$
$$\text{sum}_{HA}(a,b) = a \oplus b$$

where ab means "a and b", $a \vee b$ is "a or b" and $a \oplus b$ stands for "a exclusive-or b".

It is possible to express the logic formulas of the carries through LP_{minor}. Once implemented, the carries generated in LP_{minor} are known exactly.

Figure 3 shows the carry prediction steps for a 4-bit multiplication with one column in LP_{major} (a).

- (b) The first step computes the carries generated in the least significant column of LP_{minor}, col_8. Since there is only one partial product there, no carry can be generated.
- (c) The carry generated in col_7 is added to col_6. This carry is expressed logically as $x_0x_1y_0y_1$.
- (d) The first carry in col_6, $A(x,y) = \text{carry}_{FA}(x_0y_2, x_1y_1, x_2y_0)$ is similarly added to col_5, and the corresponding sum bit $B(x,y) = \text{sum}_{FA}(x_0y_2, x_1y_1, x_2y_0)$ replaces the three partial products in col_6.
- (e) Finally $x_0x_1y_0y_1$ and $B(x,y)$ are removed and their carry $C(x,y)$ is added to col_5, in LP_{major}.

We note LP'_{major} and LP'_{minor} the parts of the PPA corresponding to LP_{major} and LP_{minor} obtained at the end of the prediction process.

All the partial products left in LP'_{minor} are sum bits, which do not need to be computed, since every carry originally generated in LP_{minor} is now computed in LP'_{major}.

After the carry prediction process, LP'_{minor} contains only one partial product in each column. Assuming that each result bit of the multiplication has a probability $\frac{1}{2}$ of being one, the expected value of the evaluation error E_{eval} is lower than $2^{-k-1}w_{lsb}$, so that $\text{round}_{n+k}(-E_{eval}) = 0$.

We can replace $\text{round}_{n+k}(LP)$ by LP'_{major}:

$$\begin{aligned} P_{PS} &= \text{round}_n\left(MP + LP'_{major}\right) \\ &= \text{round}_n\left(MP + \text{round}_{n+k}\left(LP_{major} + LP_{minor}\right)\right) \\ &= P_{FW} \end{aligned}$$

At this point, the truncated operator we obtain gives the same result as the full-width multiplication.

B. Carry Selection

So far, the evaluation error is lower than $2^{-k-1}w_{lsb}$, but the partial products in LP'_{major} become very complicated as n grows, and a large hardware cost may result. In order to reduce the area requirements of the truncated multiplier, some carries have to be simplified. For that purpose, the logic formulas are written under their simplified disjunctive normal form, and a "filter" is applied on the last column in LP'_{major}. This consists in removing any conjunction which number of variables exceeds a given threshold t. Assuming that the input bits have a probability $\frac{1}{2}$ of being one, a conjunction of $t+1$ variables will have a probability 2^{-t-1} of being one. Applying the filter, one makes an error of about $m \times 2^{-k-t-1}w_{lsb}$, where m is the number of conjunctions that were removed in the process. The evaluation error E_{eval} is updated in accordance to this value. The multiplication result is:

$$P_{PS} = \text{round}_n\left(MP + LP'_{major} + \text{round}_{n+k}(-E_{eval})\right) \quad (2)$$

Let us consider the previous example (Figure 3) of an n-bit truncated multiplication with $n = 4$ and $k = 1$. The only column in LP'_{major} contains $x_3y_0, \cdots, x_0y_3$, $A(x,y)$ and $C(x,y)$, where $C(x,y) = x_0x_1x_2y_0y_1y_2 \vee x_0x_1\overline{x_2}y_0y_1\overline{y_2}$ in its disjunctive normal form.

If no threshold is imposed, the truncated multiplication is equivalent to the ideal rounded multiplication.

For a value of the threshold $t = 4$, both conjunctions in $C(x,y)$ are removed, and the evaluation error is increased by the probability of $C(x,y)$ being one.

If the restriction on t is set down to 2, the three terms constituting $A(x,y)$ will also disappear. The evaluation error is further increased by the probability of $A(x,y)$ being one, and the truncated multiplier is now equivalent to a CCT multiplier.

Multiplication scheme	$\lvert\beta\rvert$	ε_{avg}	σ	ε_{max}
Direct-truncated	7.66e-1	7.66e-1	6.19e-1	3.06
Ideal rounded	6.25e-2	2.34e-1	1.68e-1	5.00e-1
PSCT $k=1$, $t=6$	6.25e-2	2.34e-1	1.68e-1	5.00e-1
PSCT $k=1$, $t=4$	4.69e-2	2.36e-1	1.71e-1	5.62e-1
PSCT $k=1$, $t=2$	3.12e-2	2.69e-1	2.15e-1	1.06
CCT $k=1$	3.12e-2	2.69e-1	2.15e-1	1.06
VCT $k=1$	3.44e-1	3.59e-1	2.81e-1	9.37e-1

TABLE II

ERROR ANALYSIS OF SEVERAL TRUNCATED MULTIPLICATION SCHEMES FOR $n=4$

IV. RESULTS

A. Mathematical Error

For each studied multiplier scheme, the absolute bias $|\beta|$, average absolute error ε_{avg}, standard deviation σ and absolute maximum error ε_{max} are given in output lsb $w_{lsb}=2^{-n}$. They are computed exhaustively for $n\leq 8$, and using an extensive random sampling for larger values of n.

The mathematical data for the previous example is given in Table II. We can see that, by acting on the value of the threshold t, we realize a compromise between the full-width multiplier and the CCT multiplier.

B. Synthesis

We studied the implementation of truncated multiplication schemes on FPGAs. The CAD tool used was Xilinx ISE8.1i and the target was an FPGA of the Spartan 3 family (XC3S200) with a medium speedgrade (-5). Synthesis and place-and-route were area-oriented with a standard effort . The multipliers were implemented using LUTs (not hardware block multipliers).

The Xilinx devices are optimized for 4 up to 8-bit input functions. This allows us to perform an efficient implementation of PSCT multipliers with a threshold up to $t=8$. We implemented the PSCT multipliers for t running from 3 to 8. A PSCT multiplier where $t=2$ is equivalent to a CCT multiplier with the same value of the parameter k.

C. Comparisons

The comparisons are lead with some well known truncation schemes for direct multiplication, that is the CCT [7] and VCT [9] multiplications. The full-width multiplier and direct-truncated multiplier are used as a reference.

The comparisons were done for $n=8$, 12 and 16. Our method is not yet fit for higher values of n, because of the fast growing computational cost of the prediction process.

Figure 4 shows how the different schemes behave for $n=8$, 12 and 16 from top to bottom. The X-axis gives the average absolute error, which is our principal accuracy criterion. The Y-axis gives the hardware cost relatively to the full-width multiplier. The aim is to perform a good accuracy while minimizing the hardware cost. This corresponds to the lower left part of each graph.

For $n=8$, the CCT is outperformed by the PSCT for $t=3$, 4 and 5. One can compute with the same average accuracy as the CCT with smaller PSCT multipliers. Similarly, for $n=12$, the PSCT for $t=3$ requires less hardware to provide the same average accuracy as the CCT. For $n=16$, the two schemes are equivalent.

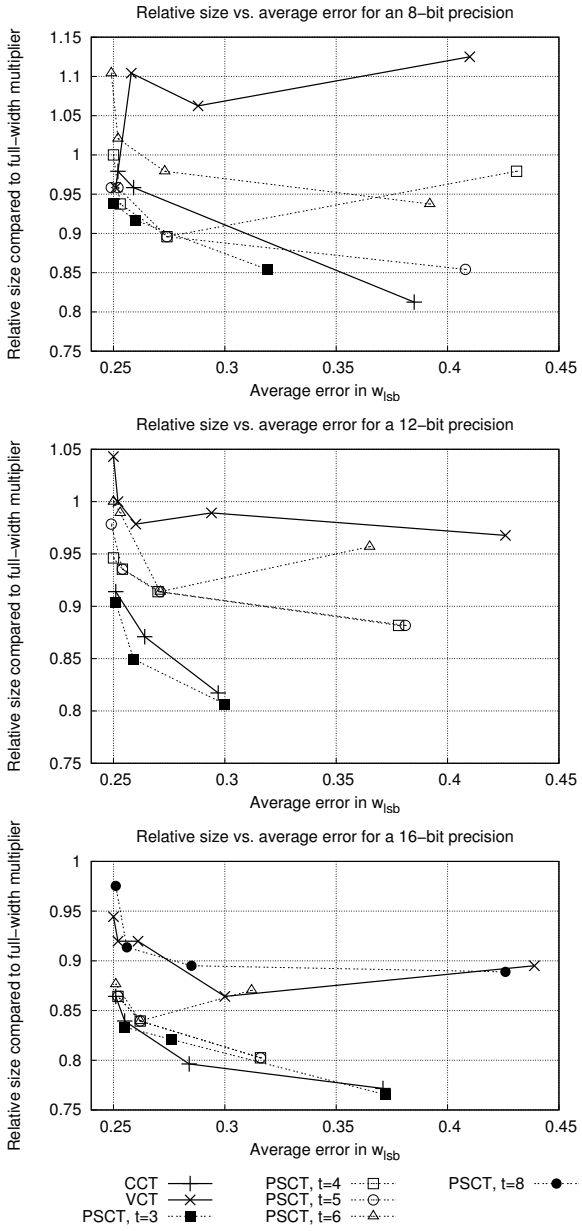


Fig. 4. Relative size vs. average error for truncated multiplication schemes.

Tables III, IV and V show accuracy results for the truncated multiplication schemes. If one wants to get an average accuracy as small as possible, that is get close to 0.25, the PSCT multiplication has a lower hardware cost than the other truncated multiplication methods.

Multiplication scheme	ε_{avg}	σ	ε_{max}	area	delay
Ideal rounded	2.49e-1	1.46e-1	5.00e-1	48	10.3
Direct-truncated	1.75	9.76e-1	7.00	32	8.9
CCT $k = 4$	2.52e-1	1.51e-1	5.66e-1	47	10.8
VCT $k = 4$	2.51e-1	1.50e-1	5.66e-1	46	11.4
PSCT $k = 4$, $t = 3$	2.50e-1	1.48e-1	6.29e-1	45	10.2
PSCT $k = 4$, $t = 5$	2.49e-1	1.47e-1	5.51e-1	46	11.7

TABLE III

ACCURACY RESULTS FOR 8-BIT TRUNCATED MULTIPLICATION SCHEMES

Multiplication scheme	ε_{avg}	σ	ε_{max}	area	delay
Ideal rounded	2.49e-1	1.45e-1	5.00e-1	93	12.7
Direct-truncated	2.73	1.24	9.00	53	11.5
CCT $k = 5$	2.51e-1	1.48e-1	5.71e-1	85	12.3
VCT $k = 4$	2.52e-1	1.49e-1	6.03e-1	93	14.6
PSCT $k = 5$, $t = 3$	2.51e-1	1.48e-1	6.56e-1	84	12.3
PSCT $k = 5$, $t = 4$	2.50e-1	1.46e-1	5.94e-1	88	14.3
PSCT $k = 5$, $t = 5$	2.49e-1	1.46e-1	5.78e-1	91	14.3

TABLE IV

ACCURACY RESULTS FOR 12-BIT TRUNCATED MULTIPLICATION SCHEMES

Multiplication scheme	ε_{avg}	σ	ε_{max}	area	delay
Ideal rounded	2.49e-1	1.45e-1	5.00e-1	162	15.2
Direct-truncated	3.76	1.48	12.5	95	13.2
CCT $k = 6$	2.51e-1	1.47e-1	5.62e-1	140	15.0
VCT $k = 4$	2.52e-1	1.50e-1	6.36e-1	149	15.3
VCT $k = 5$	2.50e-1	1.46e-1	5.59e-1	153	15.3
PSCT $k = 5$, $t = 4$	2.52e-1	1.50e-1	6.41e-1	140	16.0
PSCT $k = 5$, $t = 6$	2.51e-1	1.49e-1	6.26e-1	142	18.2

TABLE V

ACCURACY RESULTS FOR 16-BIT TRUNCATED MULTIPLICATION SCHEMES

CONCLUSION

We presented a new truncated multiplication scheme. The method first computes the logic expression of the carries propagated from LP_{minor}, then performs simplifications while keeping control over the introduced error. This scheme achieves an improvement both for accuracy and hardware requirements over previous schemes. The proposed method has been implemented on FPGAs, it shows an area reduction for comparable accuracy on 8 and 12-bit multipliers.

In a near future we plan to improve the speed of our method in order to deal with larger multipliers. We also plan to study the effects of different groupings of the partial products during the carry prediction phase, that should lead to accuracy and hardware cost improvements.

REFERENCES

[1] E. G. Walters and M. J. Schulte, "Efficient function approximation using truncated multipliers and squarers," in *Proc. of the 17th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, June 2005, pp. 232–239.

[2] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[3] Y. C. Lim, "Single-precision multiplier with reduced circuit complexity for signal processing applications," *IEEE Transactions on Computers*, vol. 41, no. 10, pp. 1333–1336, Oct. 1992.

[4] T.-B. Juang and S.-F. Hsiao, "Low-error carry-free fixed-width multipliers with low-cost compensation circuits," *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, vol. 52, no. 6, pp. 299–303, June 2005.

[5] L.-D. Van and C.-C. Yang, "Generalized low-error area-efficient fixed-width multipliers," *IEEE Transactions on Circuits and Systems—I: Regular Papers*, vol. 52, no. 8, pp. 1608–1619, Aug. 2005.

[6] S. S. Kidambi, F. El-Guibaly, and A. Antoniou, "Area-efficient multipliers for digital signal processing applications," *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, vol. 43, no. 2, pp. 90–95, Feb. 1996.

[7] M. J. Schulte and E. E. Swartzlander, "Truncated multiplication with correction constant," in *IEEE Workshop on VLSI Signal Proccessing VI*, Oct. 1993, pp. 388–396.

[8] K.-J. Cho, K.-C. Lee, J.-G. Chung, and K. K. Parhi, "Design of low-error fixed-width modified booth multiplier," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 5, pp. 522–531, May 2004.

[9] E. J. King and E. E. Swartzlander, "Data-dependent truncation scheme for parallel multipliers," in *Proc. of 31th Asilomar Conference on Signals, Systems & Computers*, vol. 2. IEEE, Nov. 1997, pp. 1178–1182.

[10] E. E. Swartzlander, "Truncated multiplication with approximate rounding," in *Proc. of 33th Asilomar Conference on Signals, Systems & Computers*, vol. 2. IEEE, Oct. 1999, pp. 1480–1483.

[11] J. E. Stine and O. M. Duverne, "Variations on truncated multiplication," in *Proc. Euromicro Symposium on Digital System Design (DSD)*. IEEE, Sept. 2003, pp. 112–119.

Small FPGA polynomial approximations with 3-bit coefficients and low-precision estimations of the powers of x

Romain Michard, Arnaud Tisserand and Nicolas Veyrat-Charvillon
Arénaire project (CNRS–ENS Lyon–INRIA–UCBL), LIP
Ecole Normale Supérieure de Lyon (ENS Lyon)
46 allée d'Italie. F–69364 Lyon, France
E-mail: {firstame.lastname}@ens-lyon.fr

Abstract

This paper presents small FPGA implementations of low-precision polynomial approximations of functions without multipliers. Our method uses degree-2 or degree-3 polynomial approximations with at most 3-bit coefficients and low-precision estimations of the powers of x. Here we denote by 3-bit coefficients values with at most 3 non-zero and possibly non-contiguous signed bits (e.g., $1.001000\overline{1}$). This leads to very small operators by replacing the costly multipliers by a small number of additions. Our method provides approximations with very low average error and is suitable for signal processing applications.

1. Introduction

In digital systems, polynomial approximations are widely used. The elementary functions (e.g., sine, cosine, logarithm, exponential), for instance, are often evaluated using polynomials [7]. Algebraic functions, such as square root or reciprocal square root can be efficiently approximated using polynomials. Low-degree polynomials are often used for evaluating reciprocals in digital signal processing applications such as frequency demodulation.

The size of the multipliers is often a problem when implementing function approximations in hardware. Several solutions have been investigated to limit their size. Methods based on tables and small multiplications are often used [12, 8, 4, 2]. For small precision, some methods, such as the multipartite tables, have been introduced to avoid the use of multipliers [11, 10, 1]. The partial product arrays (PPAs) method [5] uses converging series where all the operations have been developed at the bit level and where the low-weight terms are discarded.

In this work we focus on polynomial approximations with at most 3 non-zero bits coefficients and low-precision estimations of the powers of x. We deal with degree-2 or degree-3 polynomial approximations: $P(x) = p_0 + p_1 x + p_2 x^2 [+p_3 x^3]$. The coefficients p_1, p_2 and p_3 are represented using at most 3 non-zero signed bits in order to replace the multipliers by a small number of additions. The coefficient p_0 is kept as large as possible since it is an additive term. In order to further decrease the size of the operators, we use low-precision estimations of the powers of x (i.e., x^2 and x^3). The proposed method leads to approximations with a maximum error limited to a few LSBs, but with very low average error. The obtained average error is close to the error of the minimax polynomial. This makes our method an attractive solution for some signal processing applications.

This paper is organized as follows. The notations and background are presented in Section 2. Our contribution is presented in Section 3. Section 4 presents the FPGA implementations. We compare our results with other methods in Section 5. We conclude in Section 6.

2. Notations and Minimax Polynomial Approximations

In this work, we deal with the evaluation of a function f with inputs and outputs in fixed-point format. The argument x is in the domain $[a, b[$ and the result $f(x)$ is in the range $[a', b'[$ (or $]a', b']$). Our work can be straightforwardly extended to other forms of intervals (e.g., $[a, b]$). The integer d denotes the degree of the polynomials. The argument x is a w_I-bit number and the output $f(x)$ is a w_O-bit number. The notation $()_2$ denotes the binary representation of a value. For instance the value 3.125 is represented in binary by $(11.001)_2$. The quantified coefficients of the polynomial will be represented in the *borrow-save* format [3]. Bits with a negative weight are denoted by $\overline{1}$.

The input argument x is considered as exact. The *approximation error* measures the distance between the mathematical function f and the approximated function used to evaluate it. The *rounding error* due to the discrete nature of

Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)
1063-6862/05 $20.00 © 2005 IEEE

the final and intermediate values adds up to the approximation error. In order to limit the rounding error, we introduce g additional *guard bits* for the intermediate computations (i.e., they are done on words of $w_O + g$ bits).

In order to measure the theoretical approximation error ϵ_{th} due to the use of the polynomial P to evaluate the function f on $[a, b]$, we use the distance (estimated using the Maple `infnorm` function):

$$\epsilon_{th} = ||f - P||_\infty = \max_{a \le x \le b} |f(x) - P(x)|.$$

For a given argument x it is possible to evaluate the effective total error $\epsilon = f(x) - \texttt{output}(P(x))$ which includes all kinds of error. Here, $\texttt{output}(P(x))$ is the result of the evaluation of $P(x)$ by the circuit using finite precision computations. As the proposed method targets low-precision approximations (up to 16 bits), the *average* total error ϵ_{avg}, its *standard deviation* σ and the *maximum* error ϵ_{max} can be computed. Indeed, for all possible values of x, the effective result is evaluated and compared to the theoretical value.

The polynomial approximations used in the following are based on the *minimax* polynomial approximation as a starting point. The degree-d minimax polynomial approximation to f on $[a, b]$ is the polynomial P^* that satisfies:

$$||f - P^*||_\infty = \min_{P \in \mathcal{P}_d} ||f - P||_\infty,$$

where $\mathcal{P}_d$ is the set of polynomials with real coefficients and degree at most d. Minimax approximations can be computed thanks to an algorithm due to Remes [9].

Example 1 *Degree-3 minimax approximation of the sine function on* $[0, \pi/4]$ *(results from Maple truncated to* 10 *decimals),* $\epsilon_{th} = 0.0000474552$ *(i.e.,* 14.3 *bits of accuracy):*

$$P(x) = -0.0000474552 + 1.0017332478\,x - 0.0095826177\,x^2 - 0.1522099691\,x^3.$$

In the following, error is also expressed in number of correct bits. For instance, the error $\epsilon_{th} = 2.3 \times 10^{-3}$ is equivalent to 8.7 correct bits (CB).

3. The 3-Bit Coefficients and Estimations of Powers of x Method

The proposed method is based on the following steps:

Step 1 determination of the minimax polynomial approximation P_{th} (done using `minimax` Maple function);

Step 2 quantification of the coefficients of P_{th} to 3 non-zero bits, this gives polynomial P_q;

Step 3 estimation of the powers of x in polynomial P_q;

Step 4 fine tuning of the coefficients, this gives the polynomial P_t.

3.1. 3-bit Quantification of the Coefficients

In order to replace large multipliers by a small number of additions, the coefficients (p_1, p_2 and p_3 if any) of polynomial P_{th} are quantified to values with at most 3 non-zero bits. The constant coefficient p_0 is not quantified. It is kept as large as possible since it is an additive term.

The quantification of coefficient p_i with NZ non-zero bits (here $NZ \le 3$) and a relative accuracy of 2^{-k} (the span of the NZ bits is at most k-bit wide) is obtained using an iterative algorithm. At each iteration, the power of 2 the nearest to p_i is determined and subtracted to p_i. This iteration is applied to the remainder until NZ non-zero bits are used or an accuracy less than 2^{-k} is reached (when the remainder is zero or the difference of ranks between the most significant and the least significant non-zero bits is greater than k).

The result of the quantification step is one of the possible representations of p_i with at most NZ non-zero bits and an accuracy less than or equal to 2^{-k}. The returned quantified coefficient also has the smallest possible span (the difference between the ranks of the most significant and least significant non-zero bits). For instance, the algorithm favors the case $(11.01)_2$ rather than $(10\overline{1}.01)_2$ for the value 3.25.

Depending on the target, one specific kind of representation may be preferable. For instance, for ASIC implementations, it may be more efficient to favor the representation with the smallest number of negative bits (a subtraction may be slightly larger than an addition). In this case, the quantification of the value 1.375 with the representation $(1.011)_2$ should be preferred to the equivalent representation $(1.10\overline{1})_2$.

Example 2 *Quantification of the value* $v = 1.0017332478$ *with at most* 3 *non-zero bits for several spans* k*, the quantified value is denoted* v_q*:*

- $k = 8$, $v_q = 2^0 = (1.0000000)_2$, $v - v_q = 0.0017332478$ *(i.e.,* 9.1 *CB);*
- $k = 10$, $v_q = 2^0 + 2^{-9} = (1.000000001)_2$, $v - v_q = -0.0002198772$ *(i.e.,* 12.1 *CB);*
- $k = 13$, $v_q = 2^0 + 2^{-9} - 2^{-12} = (1.000000001001\overline{1})_2$, $v - v_q = 0.0000242634$ *(i.e.,* 15.3 *CB).*

Figure 1 presents the quantification error for all the 10-bit values of x in $[0, 1[$ quantified to at most 3 non-zero bits with a maximum span of $k = 8$ bits. The average error is 0.0000534057 (i.e., 14.1 bits of accuracy), its standard deviation is 0.0031827562. The maximum error is 0.015625 (i.e., 4 LSBs).

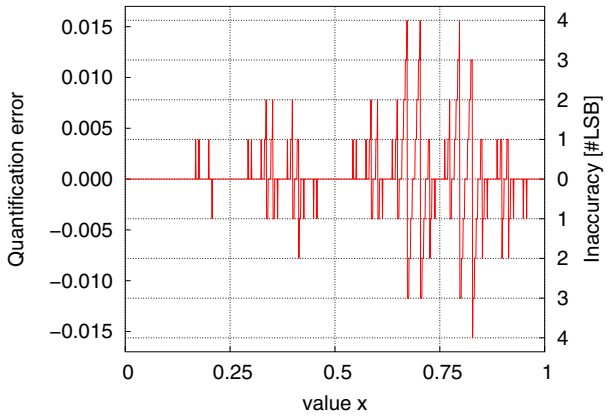


Figure 1. Quantification error for 10-bit values in $[0, 1[$ with $NZ \leq 3$ and $k = 8$.

3.2. Low-Precision Estimations of the Powers of x

Once the coefficients of the polynomial have been quantified to 3-bit values, some multiplications remain during the evaluation of $P(x)$. Indeed the square x^2 and the cube x^3 of the argument have to be computed. In this work we also try to replace these "multiplications" by a small number of additions. For that, we replace the values of x^2 and x^3 by estimations of these values.

First of all, we apply the standard simplifications for the computations of the partial products of x^2 and x^3. Those simplifications can be found in [3, 6].

The estimation is done by taking into account only the c first columns of the partial products. The number of columns c is a parameter in the exploration space.

Table 1 presents the impact of the number of columns c in the estimation of x^2 on the evaluation of $p_2 \times x^2$ with $p_2 = 0.1882871881$, $w_I = k = 8$ bits and $NZ \leq 3$. For each value of c, several values are reported: the number of partial products $\#_{pp}$, the average error ϵ_{avg}, its standard deviation σ and the maximum error ϵ_{max}. Those error characteristics are evaluated for the 2^8 possible values of x in $[0, \pi/4[$. The number of partial products when all the columns are used in x^2 is 36 and not $64 = 8^2$ due to the simplifications from [3, 6].

The values of the average error reported in Table 1 show that the loss of accuracy in the computation of $p_2 \times x^2$ may be large when c is small. One can conclude that it is not a good idea to estimate the powers of x. This is false! Table 2 presents the same study for the *complete* computation of the sine function on $[0, \pi/4[$. The polynomial used with the parameters $w_I = w_O = k = 8$, $g = 2$ and $NZ \leq 3$ is:

$$-(0.000000001)_2 + (1.0001001\overline{1})_2 x - (0.0011000001)_2 x^2.$$

c	$\#_{pp}$	ϵ_{avg}	σ	ϵ_{max}
4	8	0.78×10^{-2}	0.58×10^{-2}	0.23×10^{-1}
5	12	0.59×10^{-2}	0.42×10^{-2}	0.18×10^{-1}
6	16	0.28×10^{-2}	0.19×10^{-2}	0.88×10^{-2}
7	20	0.18×10^{-2}	0.11×10^{-2}	0.52×10^{-2}
all	36	0.13×10^{-2}	0.67×10^{-3}	0.32×10^{-2}

Table 1. Size and accuracy of the evaluation of $p_2 \times x^2$ for several values of c.

For this polynomial several values of c have been tested for the estimation of x^2. The error characteristics are evaluated for the 2^8 possible values of x in $[0, \pi/4[$. Table 2 clearly shows that a rough estimation of x^2 is sufficient to provide a correct average error and maximum error. For this example, approximations with only $c = 5$ or 6 columns (for x^2) have an accuracy equivalent to the solution with the complete, and costly, computation of x^2.

c	ϵ_{avg}	σ	ϵ_{max}
4	0.44×10^{-2}	0.35×10^{-2}	0.15×10^{-1}
5	0.23×10^{-2}	0.16×10^{-2}	0.75×10^{-2}
6	0.22×10^{-2}	0.15×10^{-2}	0.75×10^{-2}
7	0.21×10^{-2}	0.15×10^{-2}	0.75×10^{-2}
all	0.21×10^{-2}	0.15×10^{-2}	0.75×10^{-2}

Table 2. Accuracy of the evaluation of $sin(x)$ on $[0, \pi/4[$ for several values of c.

3.3. Fine Tuning of the p_i's

Due to the estimation of the powers of x, the overall accuracy can be slightly improved by modifying the value of the quantified coefficients. Indeed, using only the c most significant columns in the partial products of x^i leads to underestimate its actual value. In order to compensate this underestimation, one can try to modify the coefficient p_i.

The fine tuning algorithm is quite simple. For each monomial $p_i x^i$, it determines its average error ϵ_i. If ϵ_i is less than zero, 1 is added to the LSB of the quantified coefficient p_i (-1, if $\epsilon_i > 0$). This step is repeated while the accuracy of the result is improved. The result of the fine tuning step depends on the actual value of c.

Example 3 *Fine tuning of the sine function on $[0, \pi/4[$ with $w_I = w_O = 12$, $NZ \leq 3$ and $g = 2$. The estimation of x^2 is done with $c = 4$ and $c = 8$ for x^3. Before the fine tuning, $\epsilon_{avg} = 0.91 \times 10^{-3}$ (i.e., 9.9 CB). Fine tuning on the individual coefficients:*

- $p_1 = 2^0 + 2^{-9} - 2^{-12}$ *not modified;*
- $p_2 = -2^{-7} - 2^{-9} + 2^{-13}$ *modified to* $-2^{-7} - 2^{-9}$*;*
- $p_3 = -2^{-3} - 2^{-5} + 2^{-8}$ *modified to* $-2^{-3} - 2^{-5}$.

After the fine tuning, $\epsilon_{avg} = 0.72 \times 10^{-3}$ *(i.e.,* 10.4 *CB) and two additions have been suppressed.*

3.4. A Complete Example

Let us approximate the sine function on $[0, \pi/4[$ with 8-bit input and output ($w_I = w_O = 8$) with a degree-2 polynomial. The corresponding minimax polynomial $P_{th}(x)$ is:

$$-0.0023098047 + 1.0540785973\,x - 0.1882871881\,x^2.$$

The quantification step is performed with the parameters $k = 8$, $NZ \leq 3$ and $g = 2$. The result of the quantification step is the polynomial $P_q(x)$:

$$-(0.000000001)_2 + (1.000100\overline{1})_2\,x - (0.0011000001)_2\,x^2.$$

The next step is the estimation of x^2. Here we only use $c = 3$ columns. The value of the coefficients of $P_q(x)$ does not change during this step.

The last step is the fine tuning of the coefficient p_2. By chance, the number of non-zero bits of the new coefficient is decreased. The result is the polynomial $P_t(x)$:

$$-(0.000000001)_2 + (1.000100\overline{1})_2\,x - (0.0100\overline{1})_2\,x^2.$$

The accuracy measured for each step is reported in Table 3. The results show that the 3-bit quantification is not the main source of error. In this case, the very rough estimation of x^2 (only 3 columns) leads to an average accuracy of 7 bits. After the fine tuning step, the average error is slightly improved. These results show that even with very "cheap" estimation of the powers of x, reasonable average accuracy can be achieved using our method. Figure 2 presents an illustration of the overall computation for this example.

Step	ϵ_{avg}	ϵ_{max}	Cost
Minimax	$\epsilon_{th} = 0.23 \times 10^{-2}$		$3\times, 2\pm$
Quantif.	0.16×10^{-2}	0.53×10^{-2}	$1\times, 2\pm$
x^2 estim.	0.69×10^{-2}	0.23×10^{-1}	$7\pm$
Fine tuning	0.41×10^{-2}	0.18×10^{-1}	$6\pm$

Table 3. Accuracy and cost evolution for the different steps for $sin(x)$ **on** $[0, \pi/4[$**.**

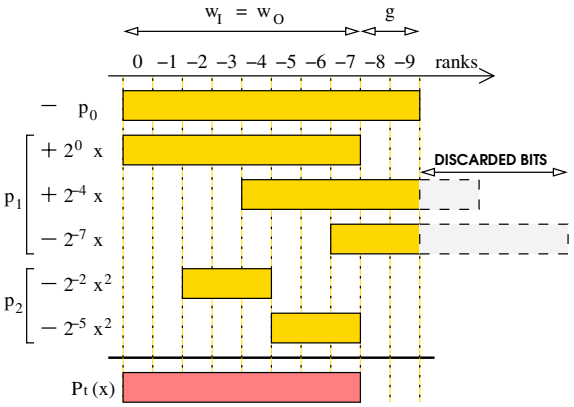


Figure 2. Illustration of the computation of P_t **for** $sin(x)$ **on** $[0, \pi/4[$**.**

3.5. Some Accuracy Results

Table 4 summarizes the accuracy results for the sine function. For degree-3 approximations, the numbers of columns are given in the order x^2, x^3. These results show that correct approximations can be achieved using rough estimations of x^2 and x^3. Table 5 presents accuracy results for other common functions. For function 2^x with a degree-3 polynomial, using only $NZ \leq 3$ for coefficient p_1 leads to a very small accuracy. This can be improved using $NZ = 4$ for this specific coefficient (line $d = 3^\star$ in Table 5).

Target	c	ϵ_{avg}	ϵ_{max}
$w_I = 8$	4	0.43×10^{-2}	0.14×10^{-1}
$d = 2$	5	0.24×10^{-2}	0.09×10^{-1}
	6	0.18×10^{-2}	0.05×10^{-1}
	all	0.16×10^{-2}	0.05×10^{-2}
$w_I = 12$	2,8	0.61×10^{-3}	0.31×10^{-2}
$d = 3$	3,10	0.44×10^{-3}	0.24×10^{-2}
	4,11	0.38×10^{-3}	0.16×10^{-2}
	5,12	0.17×10^{-3}	0.08×10^{-2}
	all	0.08×10^{-3}	0.03×10^{-2}

Table 4. Accuracy results for $sin(x)$ **on** $[0, \pi/4[$**,** $w_I = w_O = k$**,** $NZ \leq 3$**,**$g = 2$**.**

4. FPGA Implementations

The implementation results have been obtained using Xilinx Virtex-E XCV400E FPGA with the ISE 5.2.03i environment. Standard effort has been used both for synthesis

f	d	w_I	c	ϵ_{avg}	ϵ_{max}
$1/x$	2	8	5	0.39×10^{-2}	0.24×10^{-1}
$[1, 2[$	3	12	9,9	0.10×10^{-2}	0.36×10^{-2}
2^x	2	8	6	0.37×10^{-2}	0.11×10^{-1}
$[0, 1[$	3	12	6,6	0.68×10^{-2}	0.21×10^{-1}
	$3^\star$	12	6,6	0.19×10^{-2}	0.08×10^{-1}
$\sqrt{x}$	2	8	5	0.15×10^{-2}	0.52×10^{-2}
$[1, 2[$	3	12	7,7	0.21×10^{-3}	0.98×10^{-3}
$1/\sqrt{x}$	2	8	5	0.32×10^{-2}	0.11×10^{-1}
$[1, 2[$	3	12	7,7	0.89×10^{-3}	0.40×10^{-2}

Table 5. Accuracy results for other functions.

and place-and-route (P&R) steps. The reported results are post-P&R values. Area and period are reported in number of slices and in nanoseconds respectively.

The implementation of our method in FPGAs is very simple. The few partial products of the powers of x are computed in parallel with the computation of the sum of p_0 and the 3 terms of $p_1 \times x$. This sum is then added to the 3 terms of $p_2 \times x^2$ and $p_3 \times x^3$ if any. The critical path is a chain of several adders which can be easily pipelined.

Table 6 presents the results of the implementation of different versions of multiplier-based polynomiers for several size and degree parameters. Those versions are:

- generic : full-width non-constant coefficients, multipliers for $p_i \times x^i$ and optimized multipliers for x^i;
- constant : full-width but constant coefficients (for sine function on $[0, \pi/4[$), constant multipliers for $p_i \times x^i$ and optimized multipliers for x^i;
- quantified : quantified coefficients (for sine function on $[0, \pi/4[$, $k = w_I$ and $NZ \leq 3$), additions for $p_i \times x^i$ and optimized multipliers for x^i;

Table 7 presents the implementation results for polynomial approximations of sine function on $[0, \pi/4[$ with coefficients quantified to 3-bit values and estimation of x^2 or x^3. These fully optimized results show significant improvements both for speed and area. Table 8 presents results for other functions and parameters.

5. Comparison with Previous Work

We compare our results with two other methods dedicated to this range of precision: the multipartite tables from [1] and the single multiplication second order method (SMSO) from [2]. Those two references have been used because they provide implementation results for FPGAs. Our

Version	d	2		3	
	w_I, w_O	12	16	12	16
Generic (w. large mult.)	Area	235	411	895	1886
	Period	35.9	37.9	54.6	50.4
Constant (w. cst. mult.)	Area	107	192	697	1570
	Period	27.9	29.9	52.3	59.0
Quantified (w. cst. mult.)	Area	85	136	651	1475
	Period	21.3	22.7	35.5	38.5

Table 6. FPGA implementation results for generic polynomiers (using multipliers) for $\sin(x)$ on $[0, \pi/4[$.

Version	d	2		3	
	w_I, w_O	12	16	12	16
$c = 6$	Area	50	61	78	99
	Period	18.7	19.5	23.7	22.9
$c = 4$	Area	46	57	61	82
	Period	18.6	18.9	20.1	22.1
$c = 3$	Area	45	56	50	71
	Period	18.2	18.5	20.5	23.5

Table 7. FPGA implementation results for polynomiers with 3-bit coefficients and estimations of the powers of x for $\sin(x)$ on $[0, \pi/4[$.

results in Table 9 show smaller circuits. The comparison at the speed level is more complex because the target FPGAs in our work and [1] (Virtex-E) are slower than those in [2] (Virtex-II). In practice, on comparable FPGA devices, our method may lead to faster operators since the size is significantly smaller. Our results are attractive compared to these methods but one should keep in mind that the accuracy targets are not the same. Methods from [1] and [2] provide faithful rounding while ours provides a very small average error but no faithful rounding. So, our method is suitable for some applications in signal processing.

An interesting method to compare with is the partial product arrays from [5]. Unfortunately, it seems that there is no FPGA implementation of this method. Shift-and-add algorithms, such as CORDIC [7], are not interesting for such low-precision implementations.

6. Conclusion and Future Prospects

We have presented a method for low-precision approximation of functions in hardware. The presented method

f	$\sin(x)$ on $[0, \pi/4[$		$\sqrt{(x)}$ on $[1, 2[$	
Degree	2	3	2	3
$w_I = w_O$	8	12	8	12
c	5	$3, 10$	5	$7, 7$
Area [# slices]	27	141	17	86
Period [ns]	16.7	28.5	17.2	29.4

Table 8. FPGA implementation results for other parameters or functions.

	Multipartite [1]		SMSO [2]		Ours	
w_I	Area	Period	Area	Period	Area	Period
8	19	16.6	21	8	27	14.9
12	76	18.0	63	14	50	20.5
16	280	24.8	123	19	71	23.5

Table 9. Comparison with methods from [1] and [2] for $\sin(x)$ on $[0, \pi/4[$.

leads to fast and small operators up to 16 bits of precision. The obtained operators provide very small average error with reasonable maximum error. This makes our method suitable for some applications in digital signal processing.

The proposed method is based on two modifications in the polynomial approximation. The first one is the use of quantified coefficients with up to 3 non-zero bits instead of full width coefficients. The second one is the use of estimations of the powers of x instead of the full width values of x^i. This leads to very small and fast circuits by replacing the costly multiplications by a small number of additions.

Compared to standard polynomiers, the proposed method shows huge improvements. Our method provides up to 40% smaller solutions than the best literature results.

In a near future, we plan to work on ASIC targets as well as higher precision (24 bits). We also plan to develop an automatic generator for our method.

Acknowledgements

This work was partially supported by an ACI grant from the French ministry of Research and Education.

References

[1] F. de Dinechin and A. Tisserand. Some improvements on multipartite tables methods. *IEEE Transactions on Computers*, 54(3):319–330, March 2005.

[2] J. Detrey and F. de Dinechin. Second order function approximation using a single multiplication on FPGAs. In *14th International Conference on Field-Programmable Logic and Applications*, pages 221–230. LNCS 3203, August 2004.

[3] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[4] M.D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, 49(7):627–637, July 2000.

[5] H. Hassler and N. Takagi. Function evaluation by table look-up and addition. In S. Knowles and W.H. McAllister, editors, *12th IEEE Symposium on Computer Arithmetic*, pages 10–16. IEEE CS, July 1995.

[6] A. A. Liddicoat and M. J. Flynn. Parallel square and cube computations. In *34th Asilomar Conference on Signals, Systems, and Computers*, pages 1325–1329. IEEE, October 2000.

[7] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[8] J. A. Pineiro, J. D. Bruguera, and J.-M. Muller. Faithful powering computation using table look-up and a fused accumulation tree. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 40–47. IEEE CS, June 2001.

[9] E. Remes. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, 198:2063–2065, 1934.

[10] M. Schulte and J. Stine. Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, 48(8):842–847, August 1999.

[11] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson, and C. R. Role. CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. *IEEE Journal of Solid State Circuit*, 19(4):497–506, August 1984.

[12] N. Takagi. Powering by a table look-up and a multiplication with operand modification. *IEEE Transactions on Computers*, 47(11):1216–1222, 1998.

Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales

Romain Michard[1], Arnaud Tisserand[2] et Nicolas Veyrat-Charvillon[1]

[1] Arénaire, LIP, CNRS–ENS Lyon–INRIA–UCB Lyon
46 allée d'Italie. F-69364 Lyon
romain.michard@ens-lyon.fr, nicolas.veyrat-charvillon@ens-lyon.fr

[2] Arith, LIRMM, CNRS–Univ. Montpellier II
161 rue Ada. F-34392 Montpellier
arnaud.tisserand@lirmm.fr

Résumé
Cet article présente une méthode pour l'optimisation d'opérateurs arithmétiques matériels dédiés à l'évaluation de fonctions par des polynômes. La méthode, basée sur des outils récents, réduit la taille des coefficients et des valeurs intermédiaires tout en bornant l'erreur totale (approximation et évaluation). Elle conduit à des opérateurs petits et rapides tout en garantissant une bonne qualité numérique. La méthode est illustrée sur quelques exemples en FPGA.

Mots-clés : arithmétique des ordinateurs, circuit intégré numérique, approximation polynomiale, évaluation de polynôme

1. Introduction

Les systèmes sur puces embarquent de plus en plus de calculs complexes. En plus des additionneurs et multiplieurs rapides, le support matériel d'opérations évoluées est nécessaire pour certaines applications spécifiques. Des applications en traitement du signal et des images effectuent des divisions, inverses, racines carrées ou racines carrées inverses. D'autres applications utilisent des sinus, cosinus, exponentielles ou logarithmes comme la génération de signaux [4].
Différents types de méthodes sont proposés dans la littérature pour l'évaluation de fonctions. On trouve dans [5] les bases d'arithmétique des ordinateurs et en particulier les opérations comme la division, l'inverse, la racine carrée et la racine carrée inverse. Dans [9], les principales méthodes d'évaluation des fonctions élémentaires ($\sin$, $\cos$, $\exp$, $\log\ldots$) sont présentées.
Parmi toutes les méthodes possibles, les approximations polynomiales sont souvent utilisées. Elles permettent d'évaluer efficacement la plupart des fonctions que l'on trouve dans les applications embarquées. L'évaluation pratique d'un polynôme s'effectue en utilisant des additions et des multiplications. La surface des multiplieurs est souvent très importante dans le circuit.
En section 2, nous présentons les notations, l'état de l'art du domaine et en particulier un outil récent permettant de borner finement les erreurs. Dans ce travail, nous montrons comment obtenir des opérateurs spécifiques pour l'évaluation de fonctions par des approximations polynomiales optimisées. La méthode, proposée en section 3, intervient à deux niveaux pour réaliser cette optimisation. À partir du meilleur polynôme d'approximation théorique, nous déterminons des coefficients propices à une implantation matérielle. Ensuite, nous montrons comment réduire la taille des valeurs intermédiaires grâce à un outil récent : GAPPA [7]. Cette méthode permet de construire des approximations polynomiales à la fois efficaces et garanties numériquement à la conception. Dans la plupart des méthodes précédentes, seule l'erreur d'approximation était prise en compte pour l'optimisation. Ici, l'erreur totale (approximation et évaluation) est optimisée. Enfin, nous illustrons en section 4 l'impact de nos optimisations pour quelques exemples de fonctions sur des circuits FPGA.

2. Notations et état de l'art

2.1. Notations

La fonction à évaluer est f pour l'argument x du domaine $[a, b]$. Toutes les valeurs sont représentées en virgule fixe. L'argument x est dans un format sur n bits tandis que le résultat $f(x)$ est sur m bits (souvent $n \approx m$). Le polynôme p utilisé pour approcher f est de degré d, ses coefficients sont notés : $p_0, p_1, \ldots, p_d$. On a donc $p(x) = \sum_{i=0}^{d} p_i x^i$. La représentation binaire des valeurs est notée $()_2$, par exemple $(11.01)_2$ représente la valeur décimale 3.25. La notation binaire signée *borrow-save* est utilisée pour les coefficients (notation en base 2 avec les chiffres $\{-1, 0, 1\}$, cf. [5]) avec le chiffre -1 noté $\overline{1}$.

2.2. Erreurs

L'argument x est considéré comme exact. L'*erreur d'approximation* ϵ_{app} est la distance entre la fonction mathématique f et le polynôme p utilisé pour approcher f. Pour déterminer cette erreur, nous utilisons la *norme infinie* définie ci-dessous et estimée numériquement par MAPLE (fonction `infnorm`).

$$\epsilon_{\text{app}} = ||f - p||_\infty = \max_{a \leq x \leq b} |f(x) - p(x)|.$$

Du fait de la précision finie des calculs, des erreurs d'arrondi se produisent au cours de l'évaluation du polynôme p en machine. L'*erreur d'évaluation* ϵ_{eval} est due à l'accumulation des erreurs d'arrondi. Même si l'erreur d'évaluation commise lors d'une seule opération est toute petite (une fraction du poids du dernier bit), l'accumulation de ces erreurs peut devenir catastrophique lors de longues séries de calculs si rien n'est prévu pour en limiter les effets.

L'erreur d'évaluation est très difficile à borner finement [8]. Le problème est lié au fait que cette erreur dépend de la valeur des opérandes. Elle est donc difficile à connaître *a priori*. Souvent, on considère le pire cas d'arrondi pour chaque opération. C'est à dire la perte de précision correspondant à la moitié du bit de poids le plus faible ($1/2$ LSB). Pour l'évaluation d'un polynôme de degré d en utilisant le schéma de Horner (d additions et d multiplications), cette borne pire cas correspond à une perte de d bits de précision. Nous allons voir dans la suite que nous pouvons faire des approximations bien moins pessimistes en utilisant l'outil GAPPA développé par G. Melquiond (cf. section 2.5).

Il est possible de mesurer l'erreur totale commise pour une valeur particulière de l'entrée x en utilisant $f(x) - \texttt{output}(p(x))$ où $\texttt{output}(p(x))$ est la sortie réelle (physique ou simulée par un simulateur au niveau bit) de l'opérateur. Cette technique est souvent utilisée pour qualifier *a posteriori* la précision d'opérateurs par des simulations exhaustives ou pour les entrées menant aux plus grandes erreurs[1]. D'une manière générale, cette technique de qualification *a posteriori* des opérateurs de calcul est coûteuse en temps de simulation et de conception si il faut modifier certains paramètres des opérateurs.

Dans la suite, nous exprimons les erreurs soit directement soit en terme de précision équivalente. Cette dernière est le nombre de bits n_c justes ou corrects équivalents à une certaine erreur ϵ avec $n_c = -\log_2 |\epsilon|$. Par exemple, une erreur de 2.3×10^{-3} est équivalente à une précision de 8.7 bits corrects.

2.3. Polynôme minimax

Il existe différents types de polynômes d'approximation [9]. Le polynôme *minimax* est, en quelque sorte, le meilleur polynôme d'approximation pour notre problème. Le polynôme minimax de degré d pour approcher f sur $[a, b]$ est le polynôme p^* qui satisfait :

$$||f - p^*||_\infty = \min_{p \in \mathcal{P}_d} ||f - p||_\infty,$$

où $\mathcal{P}_d$ est l'ensemble des polynômes à coefficients réels de degré au plus d. Ce polynôme peut être déterminé numériquement grâce à l'algorithme de Remes [10] (implanté dans la fonction MAPLE `minimax`). Le polynôme minimax est le meilleur car parmi tous les polynômes d'approximation possibles de f sur $[a, b]$, il est celui qui a la plus petite des erreurs maximales sur tout l'intervalle. On note ϵ^*_{app} l'erreur d'approximation du polynôme minimax avec $\epsilon^*_{\text{app}} = ||f - p^*||_\infty$.

Mais le polynôme minimax est un polynôme « théorique » car ses coefficients sont dans $\mathbb{R}$ et on suppose son évaluation exacte (c'est à dire faite avec une précision infinie). Pour pouvoir évaluer ce polynôme

[1] Ces entrées particulières étant très difficiles à déterminer dans le cas général.

en pratique, il faut faire deux niveaux d'approximation supplémentaires dont les erreurs peuvent se cumuler avec l'erreur d'approximation théorique ϵ^*_{app} :
– les coefficients de p^* devront être écrits en précision finie dans le format supporté par le circuit ;
– les calculs nécessaires à son évaluation seront effectués en précision finie.
Pour le moment, il n'existe pas de méthode théorique permettant d'obtenir le meilleur polynôme d'approximation en pratique (coefficients *et* évaluation dans la précision du format cible). Mais nous allons montrer dans la suite que d'importantes optimisations sont déjà possibles avec une méthode assez simple et des outils facilement accessibles.

2.4. Vers des polynômes d'approximation avec des coefficients représentables

L'un des problèmes pour l'implantation pratique d'approximations polynomiales est le fait que les coefficients du polynôme minimax ne sont pas représentables dans le format cible en précision finie. Ceci est vrai aussi pour les autres types d'approximations polynomiales [9] (Chebyshev, Legendre...).
Par exemple, prenons la fonction e^x sur l'intervalle $[1/2, 1]$ avec un polynôme de degré 2 et des coefficients sur 10 bits dont 1 en partie entière, soit $x = (x_0.x_1x_2 \dots x_9)_2$. Le polynôme minimax trouvé est[2] $1.116018832 + 0.535470996x + 1.065407154x^2$, il conduit à une précision d'approximation de 9.4 bits environ. C'est la meilleure précision atteignable avec un polynôme de degré 2 sur $[1/2, 1]$. En arrondissant au plus près les 3 coefficients de ce polynôme dans le format cible, on a alors $\frac{571}{512} + \frac{137}{256}x + \frac{545}{512}x^2$ et une précision de l'approximation de 8.1 bits. Après avoir testé tous les modes d'arrondi possibles pour chacun des coefficients dans le format cible, on trouve que le meilleur polynôme est $\frac{571}{512} + \frac{275}{512}x + \frac{545}{512}x^2$ pour une précision de l'approximation de 9.3 bits, soit un gain de 1.2 bit de précision.
Dans les dernières années, différents travaux de recherche ont été menés pour obtenir des « bons » polynômes d'approximation dont les coefficients soient exactement représentables dans le format cible.
On trouve dans [2] une méthode basée sur une formulation en un problème de programmation linéaire en nombres entiers. Les coefficients des polynômes sont représentés par des rationnels dont le dénominateur est une puissance de 2 (2^n pour un format fractionnaire sur n bits). Un polytope est construit à partir des contraintes (les tailles) sur tous les coefficients du polynôme. Chaque point du polytope est alors un polynôme dont les coefficients s'écrivent avec les tailles souhaitées. Mais seuls certains de ces polynômes sont des « bons » polynômes d'approximation de f. La liste des polynômes résultats est filtrée en deux phases. L'erreur d'approximation est mesurée en un certain nombre de points d'échantillonage pour chaque point (polynôme) possible du polytope. Le parcours du polytope donne une liste de polynômes pour lesquels la distance entre la fonction et le polynôme testé est plus petite qu'un seuil pour chacun des points d'echantillonage. Cette première liste de polynômes est ensuite traitée pour déterminer numériquement les normes infinies entre chacun des polynômes de la liste et la fonction. Cette méthode, séduisante sur le plan théorique, ne fonctionne pas encore très bien en pratique. Le temps de calcul et le volume de mémoire nécessaires sont trop importants pour le moment.
L'article présenté dans [1] utilise [2] pour produire des approximations polynomiales avec des coefficients creux (avec beaucoup de 0) pour des petits degrés (3 ou 4). Toutefois, du fait des énormes besoins en temps de calcul et de taille mémoire de [2], les approximations trouvées sont limitées à une douzaine de bits de précision. De plus, seule l'erreur d'approximation est prise en compte.

2.5. Outils pour le calcul de bornes d'erreur globale

L'étude de méthodes pour borner finement les erreurs de calcul est un domaine de recherche actif depuis quelques années. Dans le domaine du traitement du signal, des méthodes assimilant les erreurs d'arrondi à du bruit sont utilisées avec succès [8]. Différents outils ont été développés pour le calcul scientifique en virgule flottante. Par exemple, FLUCTUAT [6] est un analyseur statique de code qui permet de détecter certaines pertes de précision sur des programmes flottants. Le logiciel CADNA [3] implante une méthode stochastique pour l'analyse de la précision moyenne des programmes flottants (une version en virgule fixe est étudiée actuellement).
Le logiciel GAPPA [7] permet d'évaluer et de prouver des propriétés mathématiques sur des programmes numériques. La caractéristique intéressante de GAPPA dans notre problème est sa capacité à borner finement les erreurs de calcul ou à montrer que des bornes sont en dessous d'un certain seuil.
Voici un exemple simple des possibilités de GAPPA sur la fonction e^x sur $[1/2, 1]$. On suppose tous les

[2] en arrondissant les coefficients théoriques sur 10 chiffres décimaux soit envion 32 bits.

calculs effectués en virgule fixe sur 10 bits dont 1 en partie entière. Le polynôme d'approximation utilisé est $p(x) = \frac{571}{512} + \frac{275}{512}x + \frac{545}{512}x^2$, son évaluation se décrit ainsi en GAPPA :

```
1 p0 = 571/512; p1 = 275/512; p2 = 545/512;
2 x = fixed<-9,dn>(Mx);
3 x2 fixed<-9,dn>= x * x;
4 p fixed<-9,dn>= p2 * x2 + p1 * x + p0;
5 Mp = p2 * (Mx*Mx) + p1 * Mx + p0;
6 { Mx in [0.5,1] /\ |Mp-Mf| in [0,0.001385] -> |p-Mf| in ? }
```

La ligne 1 spécifie les coefficients du polynôme (choisis représentables dans le format considéré). Par convention dans la suite, les noms de variables qui commencent par un `M` majuscule représentent les valeurs mathématiques (en précision infinie), et toutes les valeurs en miniscules représentent des variables du programme (des entrées/sorties ou des registres intermédiaires). La ligne 2 indique que `x` est la version circuit de l'argument mathématique `Mx`. La construction `fixed<-9,dn>` indique que l'on travaille en virgule fixe avec le LSB de poids 2^{-9} et l'arrondi vers le bas `dn` (troncature). La ligne 3 indique que la variable `x2` est la version calculée dans le circuit de x^2. La ligne 4 décrit comment le polynôme est évalué en pratique dans le circuit tandis que la ligne 5 décrit son évaluation théorique (en précision infinie car sans l'opérateur d'arrondi `fixed<.,.>`). Enfin, la ligne 6 est la propriété cherchée (entre accolades). Les hypothèses sont en partie gauche du signe `->`, elles indiquent que la valeur mathématique de x est dans $[1/2, 1]$ et que l'erreur d'approximation entre le polynôme d'appromixation `Mp` (sans erreur d'arrondi) et la fonction mathématique `Mf` est inférieure ou égale à 0.001385 (valeur fournie par MAPLE). La partie droite du signe `->` indique que l'on demande à GAPPA de nous dire dans quel intervalle (`in ?`) est la distance entre la valeur évaluée dans le circuit du polynôme `p` et la fonction mathématique, en incluant erreurs d'approximation et d'évaluation (arrondis).
Le résultat retourné par GAPPA (version supérieure ou égale à `0.6.1`) pour ce calcul est :

```
Results for Mx in [0.5, 1] and |Mp - F| in [0, 0.001385]:
|p - F| in [0, 232010635959353905b-64 {0.0125773, 2^(-6.31303)}]
```

Après avoir répété les hypothèses, GAPPA indique qu'il a trouvé une borne pour $|p - f|$ et qu'il y au moins 6.31 bits corrects.
En modifiant les lignes 3 à 5 par les lignes suivantes, on cherche l'erreur totale commise en utilisant le schéma de Horner, GAPPA indique alors une précision globale de 6.57 bits.

```
3 y1 fixed<-9,dn>= p2 * x + p1;
4 p  fixed<-9,dn>= y1 * x + p0;
5 Mp = (p2 * Mx + p1) * Mx + p0;
```

3. Méthode d'optimisation proposée

Dans un premier temps, nous présentons rapidement la méthode, ensuite nous donnerons des détails sur chacune des différentes étapes et les informations sur les outils nécessaires.
Les données nécessaires en entrée de la méthode sont :
– f la fonction à évaluer ;
– $[a, b]$ le domaine de l'argument x ;
– le format de l'argument x (nombre de bits n_x) ;
– μ l'erreur totale maximale cible (donnée en erreur absolue).
Les paramètres déterminés par la méthode sont :
– d le degré du polynôme à utiliser ;
– $p_0, p_1, p_2, \ldots, p_d$ les valeurs des coefficients du polynôme représentables dans le circuit ;
– n la taille utilisée pour représenter les coefficients ;
– n' la taille utilisée pour effectuer les calculs[3].

[3] Nous verrons que dans certains cas, prendre n et n' légèrement différents peut aider à limiter la taille du circuit.

Notre méthode se résume aux 3 étapes ci-dessous avec des retours possibles à une étape antérieure (rebouclage) dans certains cas :

Étape 1 : calcul du *polynôme minimax*.
On cherche p^* de degré d le plus petit possible tel que $\epsilon^*_{\text{app}} < \mu$ avec $\epsilon^*_{\text{app}} = ||f - p^*||_\infty$. Cette première phase donne l'erreur d'approximation ϵ^*_{app} minimale atteignable en supposant tous les calculs faits et coefficients représentés en précision infinie.

Étape 2 : détermination des *coefficients* du polynôme à implanter et de leur *taille*.
On cherche ici à la fois les p_i et leur taille minimale n telle que l'erreur d'approximation ϵ_{app} du polynôme p utilisé ($p(x) = \sum_{i=0}^{d} p_i x^i$) soit strictement inférieure à μ.

Étape 3 : détermination de la *taille du chemin de données*.
On cherche n' la taille minimale du chemin de données de l'étage de Horner pour effectuer les calculs. Cette dernière phase donne l'erreur d'évaluation ϵ_{eval} qui intègre les erreurs d'approximation et les erreurs d'arrondi. La valeur de n' trouvée garantit que $\epsilon_{\text{eval}} < \mu$.

Dans certains cas, il n'y a pas de solution à une étape. Il faut alors *reboucler* à l'étape précédente pour essayer d'autres solutions.
Différents types de rebouclages sont possibles. Par exemple, en fin de deuxième étape, on peut ne pas trouver des coefficients représentables pour garantir $\epsilon_{\text{app}} < \mu$. Ceci se produit lorsque la « marge » entre ϵ^*_{app} et μ était trop faible à la première étape. Il faut donc retourner à la première étape pour essayer un polynôme de degré plus grand $d \leftarrow d + 1$.
Un autre type classique de rebouclage intervient à la fin de la dernière étape. Si la taille n' trouvée est jugée trop grande devant n, il peut être intéressant de revenir à la deuxième étape pour essayer un n plus grand. Ceci permet d'avoir ϵ_{app} plus petit et donc plus de marge pour les erreurs d'arrondi.
Dans les descriptions données jusqu'ici, nous utilisons des contraintes du style $\epsilon_{\text{app}} < \mu$ et pas $\epsilon_{\text{app}} \leq \mu$. En pratique, il faut de la marge entre ϵ^*_{app} et μ puis entre ϵ_{app} et μ. Le caractère stricte des contraintes pour les étapes 1 et 2 est donc nécessaire. Pour la dernière étape, peut être pouvons-nous trouver une fonction (probablement triviale et donc peu intéressante) telle que l'on ait $\epsilon_{\text{eval}} = \mu$ à la fin. Cela nous semble très improbable. De plus, l'utilisateur saurait traiter convenablement ce cas.

3.1. Calcul du polynôme minimax

Dans cette étape on utilise la fonction `minimax` de MAPLE. On commence avec $d = 1$, et on incrémente d jusqu'à ce que le polynôme minimax p^* trouvé soit tel que $\epsilon^*_{\text{app}} < \mu$.
Voici un exemple pour $f = \log_2(x)$ avec x dans $[1, 2]$:

```
1 > minimax(log[2](x),x=1..2,[1,0],1,'err'); -log[2](err);
2     -.9570001094+1.000000000*x
3     4.537124583
4 > minimax(log[2](x),x=1..2,[2,0],1,'err'); -log[2](err);
5     -1.674903474+(2.024681754-.3448476634*x)*x
6     7.659796889
7 > minimax(log[2](x),x=1..2,[3,0],1,'err'); -log[2](err);
8     -2.153620718+(3.047884161+(-1.051875031+.1582487046*x)*x)*x
9     10.61615211
```

Les lignes qui commencent par un signe supérieur (« prompt » MAPLE) sont les commandes entrées par l'utilisateur. Les résultats retournés par MAPLE sont en italique. La ligne 1 signifie que l'on cherche un polynôme de degré 1 et que l'erreur d'approximation trouvée sera placée dans la variable `err`. Les lignes 2 et 3 représentent respectivement le polynôme trouvé et sa précision (en nombre de bits corrects).
Cette étape fournit trois éléments nécessaires pour la suite :
– d le degré du polynôme ;
– $p^*(x) = \sum_{i=0}^{d} p_i^* x^i$ le polynôme minimax (théorique) ;
– ϵ^*_{app} l'erreur *minimale* atteignable en utilisant p^* pour approcher f (en précision infinie).
Les deux autres étapes vont dégrader la qualité de l'approximation (i.e. fournir des erreurs plus grandes que ϵ^*_{app}). Il faut donc laisser un peu de marge entre ϵ^*_{app} et μ. Nous reviendrons sur cette marge.

Nous verrons dans l'exemple 4.2, que certains changements de variables permettent d'obtenir des coefficients d'ordres de grandeur proches entre eux. Ceci limite les recadrages en virgule fixe. Si on évalue $f(x)$ sur $[a, b]$ avec $a \neq 0$, il peut être intéressant de considérer $f(x + a)$ sur $[0, b - a]$.
Toutes les fonctions ne sont pas « approchables facilement » avec des polynômes. Nous renvoyons le lecteur aux ouvrages de références sur l'évaluation de fonctions comme [9, chap. 3] pour les fonctions élémentaires. En pratique, les fonctions usuelles s'approchent bien par des polynômes.

3.2. Détermination des coefficients du polynôme et de leur taille

Une fois p^* déterminé, il faut trouver des coefficients représentables en précision finie. On cherche n le nombre de bits du format virgule fixe des p_i tel que n soit le plus petit possible mais avec $\epsilon_{\text{app}} < \mu$.
Nous avons vu en 2.4 que le choix des coefficients est important. En arrondissant simplement les coefficients du polynôme minimax sur n bits, il est peu probable de trouver un bon polynôme d'approximation. L'exemple présenté en 2.4 montre que tester l'ensemble des combinaisons des arrondis des coefficients de p^* permet de trouver un bon polynôme. C'est ce que nous proposons de faire systématiquement dans cette deuxième phase.
Chaque coefficient p_i^* du polynôme peut être arrondi soit vers le haut $p_i = \triangle(p_i^*)$, soit vers le bas $p_i = \triangledown(p_i^*)$. Il y a 2 choix possibles par coefficient et donc 2^{d+1} combinaisons à tester au total comme illustré en figure 1. Pour chaque polynôme p, il faut déterminer $\epsilon_{\text{app}} = ||f - p||_\infty$ (fonction `infnorm` de MAPLE).

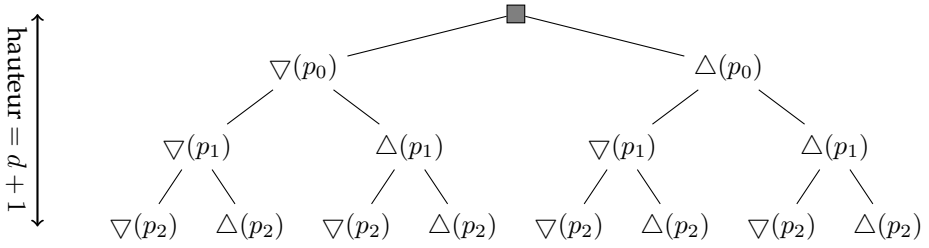


FIG. 1 – Différentes combinaisons d'arrondi pour p^* de degré $d = 2$.

Dans nos applications, d est faible ($d \leq 6$). Il y a donc au mieux quelques centaines de polynômes à tester. En pratique, chaque calcul de ϵ_{app} dure quelques fractions de seconde sur un ordinateur standard.
Nous avons fait un programme MAPLE qui teste les 2^{d+1} combinaisons d'arrondi des p_i^*. Le programme retourne la liste des polynômes candidats pour la troisième étape, c'est à dire ceux pour qui l'erreur d'approximation ϵ_{app} est minimale. On commence par $n = \lceil -\log_2 |\mu| \rceil$ (le nombre de bits correspondant à l'erreur μ), puis on test tous les arrondis des $d + 1$ coefficients sur n bits. On recommence en incrémentant n jusqu'à ce que $\epsilon_{\text{app}} < \mu$.
A cette étape, on peut aussi souhaiter modifier plus en « profondeur » certains coefficients. Par exemple, si un coefficient retourné est $0.5002441406 = (0.100000000001)_2$ et que l'on travaille sur un peu plus d'une douzaine de bits fractionnaires, il est peut être intéressant de fixer ce coefficient à 0.5 pour éliminer une opération. La multiplication par une puissance de 2 se réduit à un décalage. Nous verrons un exemple de ce genre de modification dans l'exemple 4.2. Nous ne savons pas encore formaliser simplement ce genre de traitement, mais cela constitue un de nos axes de recherche à travers l'amélioration de la méthode présentée dans [2].
Dans ce travail, nous supposons tous les coefficients représentés avec la même taille car on essaie de générer des approximations polynomiales utilisant le schéma de Horner. Dans l'avenir, nous pensons essayer de généraliser notre méthode à des tailles de coefficients différentes. Mais ceci augmente considérablement l'espace de recherche.

3.3. Détermination de la taille du chemin de données

La dernière étape permet de spécifier la taille du chemin de données pour l'évaluation suivant le schéma de Horner. Ce schéma est souvent préféré au schéma direct car il nécessite moins d'opérations et donne

souvent une erreur d'évaluation ϵ_{eval} plus faible.

$$p(x) = \begin{cases} p_0 + p_1 x + p_2 x^2 + \cdots + p_d x^d & \text{schéma direct } d \text{ add., } d + \lceil \log_2 d \rceil \text{ mult.;} \\ p_0 + x(p_1 + x(p_2 + x(\cdots + x p_d) \cdots)) & \text{schéma de Horner } d \text{ add., } d \text{ mult.} \end{cases}$$

Toutefois, dans certains cas particuliers avec des coefficients très creux, le schéma direct peut s'avérer intéressant (cf. exemple en 4.2).
L'étage de Horner permet d'évaluer $u \times v + z$. La taille du chemin de données de cet étage est n'. On commence avec $n' = n$ et on augmente n' tant que l'encadrement de l'erreur d'évaluation ϵ_{eval} par GAPPA avec n' bits pour le chemin de données ne donne pas $\epsilon_{\text{eval}} < \mu$. Pour le moment, le codage en GAPPA se fait à la main. Mais le schéma de Horner ou le schéma direct étant les mêmes aux coefficients près, nous avons les programmes GAPPA types pour lesquels il suffit de préciser les valeurs des coefficients et du degré. La boucle sur n' s'effectue en quelques minutes tout au plus sur un ordinateur standard.
La différence entre n' et n est appelée le nombre de bits de garde. Conserver n plus petit que n' permet, par exemple, de limiter la taille mémoire nécessaire pour stocker les coefficients.
Si la taille du chemin de données est un peu supérieure à n (1 à 3 bits), on pourrait revenir à l'étape 2 pour essayer un n plus grand. Notre expérience montre que le n' final change rarement. Par contre si la valeur de n' est bien plus grande que n, alors il peut être intéressant de reboucler à l'étape 2 avec un n plus grand. Ici encore, nous devons encore travailler pour formaliser ce genre de test.

3.4. Résumé de la méthode

La figure 2 résume graphiquement le fonctionnement de la méthode.

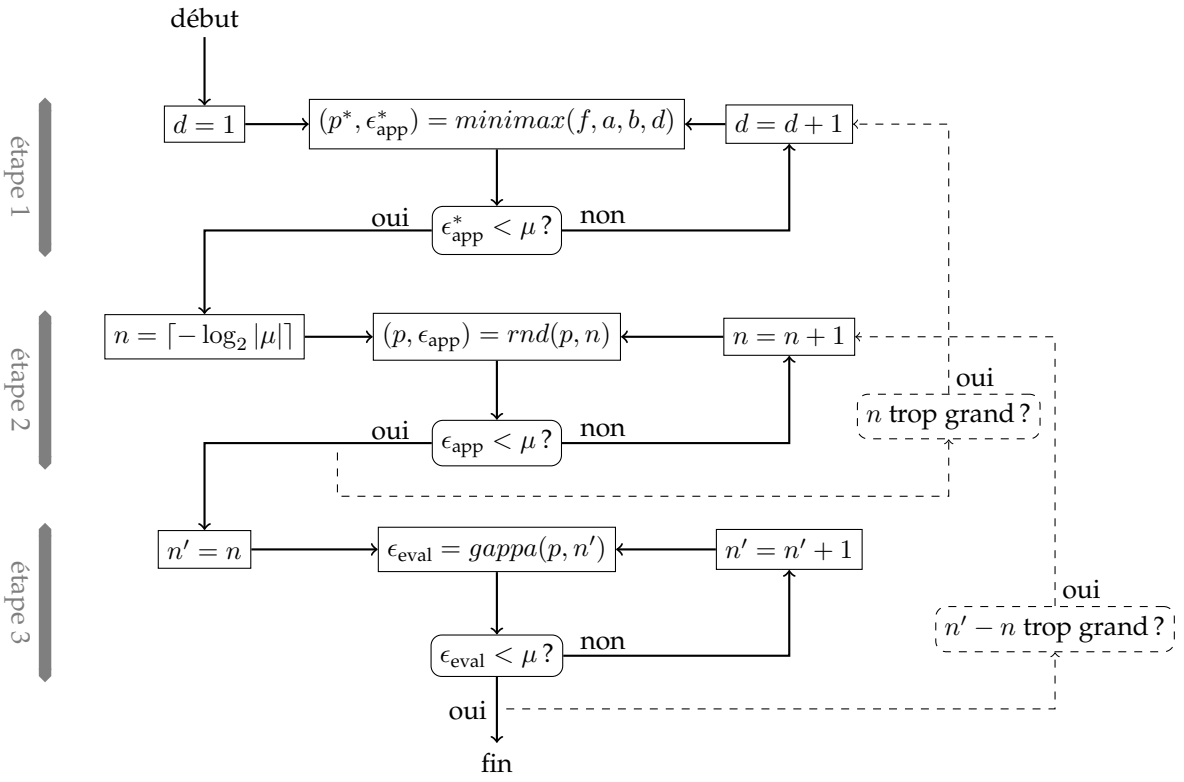


FIG. 2 – Résumé de la méthode (les rebouclages en pointillés sont facultatifs).

4. Exemples d'applications sur FPGA

Les implantations réalisées ci-dessous ont été faites pour des FPGA de la famille Virtex de Xilinx (XCV200-5) avec les outils ISE8.1i de Xilinx. La synthèse et le placement/routage utilisent une optimisation en surface avec effort élevé. Les résultats indiquent toutes les ressources nécessaires pour chaque opérateur (cellules logiques et registres fonctionnels).

4.1. Fonction 2^x sur $[0,1]$

On cherche un opérateur pour évaluer la fonction 2^x, avec x dans $[0,1]$ et une précision globale de 12 bits. La première étape est de trouver un polynôme d'approximation théorique en utilisant MAPLE. On cherche ici à minimiser le degré d du polynôme utilisé. Voici la précision d'approximation ϵ_{app} (en bits corrects) des polynômes minimax de 2^x pour d variant de 1 à 5 :

d	1	2	3	4	5
ϵ_{app}	4.53	8.65	13.18	18.04	23.15

Bien évidemment, les polynômes de degré 1 et 2 ne sont pas suffisament précis pour notre but. La solution avec le polynôme de degré 3 conduit à précision estimée dans le pire cas à 10 bits environ ($13.18-d$, car on borne à d bits la perte de précision pour un schéma de Horner de degré d) ce qui semble trop faible. De plus, les 13.18 bits corrects correspondent à l'erreur d'approximation avec le polynôme minimax avec des coefficients réels (non représentables dans le format cible).

Sans outil pour aider le concepteur, il semble donc que l'on serait obligé de choisir la solution de degré 4 ou 5. Même la solution avec un polynôme de degré 4 peut conduire à précision trop faible. Certes sa précision théorique est au moins de $18.04-4=14.04$ bits corrects mais seulement pour des coefficients en précision infinie. Le polynôme minimax de degré 4 est (les coefficients sont affichés avec 10 chiffres décimaux dans le papier, mais MAPLE est capable de les calculer avec plus de précision) : $1.0000037045 + 0.6929661227x + 0.2416384458x^2 + 0.0516903583x^3 + 0.0136976645x^4$.

Pour être certain que la solution de degré 4 peut être employée, il faut trouver un format des coefficients pour lequel le polynôme minimax avec ses coefficients arrondis dans ce format ait une précision supérieure ou égale à $12+4=16$ bits. Pour un format donné, on teste tous les modes d'arrondi possibles pour les coefficients du polynôme minimax dans le format. On trouve des polynômes acceptables à partir de 14 bits fractionnaires et 1 entier. Pour montrer que le choix de coefficients représentables est important, nous présentons ci-dessous chacune des combinaisons possibles des modes d'arrondi des coefficients et la précision d'approximation du polynôme dont les coefficients s'écrivent exactement dans le format cible. Seuls deux polynômes ont la précision souhaitée (valeurs en gras).

$(\nabla,\nabla,\nabla,\nabla,\nabla)$	12.00	$(\nabla,\nabla,\nabla,\nabla,\triangle)$	13.00	$(\nabla,\nabla,\nabla,\triangle,\nabla)$	13.00	$(\nabla,\nabla,\nabla,\triangle,\triangle)$	14.03
$(\nabla,\nabla,\triangle,\nabla,\nabla)$	13.00	$(\nabla,\nabla,\triangle,\nabla,\triangle)$	14.55	$(\nabla,\nabla,\triangle,\triangle,\nabla)$	14.99	$(\nabla,\nabla,\triangle,\triangle,\triangle)$	13.00
$(\nabla,\triangle,\nabla,\nabla,\nabla)$	13.00	$(\nabla,\triangle,\nabla,\nabla,\triangle)$	**16.13**	$(\nabla,\triangle,\nabla,\triangle,\nabla)$	**17.12**	$(\nabla,\triangle,\nabla,\triangle,\triangle)$	13.00
$(\nabla,\triangle,\triangle,\nabla,\nabla)$	15.71	$(\nabla,\triangle,\triangle,\nabla,\triangle)$	13.00	$(\nabla,\triangle,\triangle,\triangle,\nabla)$	13.00	$(\nabla,\triangle,\triangle,\triangle,\triangle)$	12.00
$(\triangle,\nabla,\nabla,\nabla,\nabla)$	13.00	$(\triangle,\nabla,\nabla,\nabla,\triangle)$	13.00	$(\triangle,\nabla,\nabla,\triangle,\nabla)$	13.00	$(\triangle,\nabla,\nabla,\triangle,\triangle)$	13.00
$(\triangle,\nabla,\triangle,\nabla,\nabla)$	13.00	$(\triangle,\nabla,\triangle,\nabla,\triangle)$	13.00	$(\triangle,\nabla,\triangle,\triangle,\nabla)$	12.99	$(\triangle,\nabla,\triangle,\triangle,\triangle)$	12.00
$(\triangle,\triangle,\nabla,\nabla,\nabla)$	12.99	$(\triangle,\triangle,\nabla,\nabla,\triangle)$	12.98	$(\triangle,\triangle,\nabla,\triangle,\nabla)$	12.91	$(\triangle,\triangle,\nabla,\triangle,\triangle)$	12.00
$(\triangle,\triangle,\triangle,\nabla,\nabla)$	12.79	$(\triangle,\triangle,\triangle,\nabla,\triangle)$	12.00	$(\triangle,\triangle,\triangle,\triangle,\nabla)$	12.00	$(\triangle,\triangle,\triangle,\triangle,\triangle)$	11.41

La solution avec le polynôme de degré 4 est donc utilisable par un bon choix des coefficients du polynôme d'approximation. Mais on va montrer que même celle de degré 3 l'est en pratique. Ce qui constitue une optimisation significative de l'opérateur mais nécessite des outils.

Le polynôme minimax de degré 3 qui approche le mieux théoriquement 2^x sur $[0,1]$ est :

$$p^*(x) = 0.9998929656 + 0.6964573949x + 0.2243383647x^2 + 0.0792042402x^3.$$

On a alors $\epsilon_{\text{app}} = ||f-p^*||_\infty = 0.0001070344$ soit 13.18 bits de précision. Ceci signifie que, quelque soit la précision des coefficients utilisés pour représenter p^* et celle utilisée pour son évaluation, on ne pourra pas avoir un opérateur avec une précision meilleure que 13.18 bits.

Afin de déterminer la version représentable de p^* que nous allons implanter, il faut trouver la taille des coefficients. Etant donné la fonction et son domaine, le format cherché est constitué d'un bit de partie

entière et $n-1$ bits de partie fractionnaire. On cherche n minimal pour une erreur d'approximation ϵ_{app} correspondante la plus proche possible du maximum théorique de 13.18.

$n-1$	12	13	14	15	16
ϵ_{app}	12.38	12.45	13.00	13.00	13.02
nb. candidats	0	0	2	2	7

En effet, pour $n-1=14$ bits, tous les modes d'arrondis possibles des coefficients donnent :

$(\nabla,\nabla,\nabla,\nabla)$	11.41	$(\nabla,\nabla,\nabla,\triangle)$	12.00	$(\nabla,\nabla,\triangle,\nabla)$	12.00	$(\nabla,\nabla,\triangle,\triangle)$	12.84
$(\nabla,\triangle,\nabla,\nabla)$	12.00	$(\nabla,\triangle,\nabla,\triangle)$	**13.00**	$(\nabla,\triangle,\triangle,\nabla)$	**13.00**	$(\nabla,\triangle,\triangle,\triangle)$	12.36
$(\triangle,\nabla,\nabla,\nabla)$	12.00	$(\triangle,\nabla,\nabla,\triangle)$	12.25	$(\triangle,\nabla,\triangle,\nabla)$	12.23	$(\triangle,\nabla,\triangle,\triangle)$	12.23
$(\triangle,\triangle,\nabla,\nabla)$	12.13	$(\triangle,\triangle,\nabla,\triangle)$	12.12	$(\triangle,\triangle,\triangle,\nabla)$	12.05	$(\triangle,\triangle,\triangle,\triangle)$	11.64

Les deux polynômes candidats sont donc :

$$\frac{8191}{8192}+\frac{2853}{4096}x+\frac{1837}{8192}x^2+\frac{649}{8192}x^3 \qquad \text{et} \qquad \frac{8191}{8192}+\frac{2853}{4096}x+\frac{919}{4096}x^2+\frac{649}{8192}x^3.$$

Tous deux conduisent à une erreur d'approximation de 0.0001220703 (13.00 bits de précision). Il reste maintenant à vérifier que l'évaluation d'au moins un de ces polynômes donne une précision finale d'au moins 12 bits. Voici ci-dessous la précision totale (approximation + évaluation) retournée par GAPPA en utilisant le schéma de Horner et le schéma direct pour évaluer $\frac{8191}{8192}+\frac{2853}{4096}x+\frac{1837}{8192}x^2+\frac{649}{8192}x^3$ pour différentes tailles du chemin de données n' :

n'	14	15	16	17	18	19	20
ϵ_{eval} Horner	11.32	11.93	12.36	12.65	12.81	12.90	12.95
ϵ_{eval} direct	11.24	11.86	12.32	12.62	12.79	12.89	12.94

Les valeurs obtenus pour l'autre polynôme ($p_2=\frac{919}{4096}$) sont équivalentes. Le schéma de Horner présente un comportement légèrement meilleur que l'évaluation directe (qui en plus est plus coûteuse en nombre d'opérations). Il faut un chemin de données sur 16 bits au moins pour obtenir un opérateur avec 12 bits de précision au final et ce à partir d'une approximation avec 13.18 bits de précision.
Passer les coefficients sur 16 bits ne modifie pas beaucoup la précision totale car on voit dans la table qui donne ϵ_{app} en fonction de n que l'on passe de 13.00 à 13.02 bits de précision seulement en passant $n-1$ de 14 à 16 pour l'approximation. Avec des coefficients et un chemin de données sur 16 bits, GAPPA indique une précision de 12.38 bits en évaluant le polynôme $p(x)=\frac{32765}{32768}+\frac{22821}{32768}x+\frac{7351}{32768}x^2+\frac{649}{8192}x^3$.
Deux solutions ont été implantées pour cet opérateur : celle optimisée (degré 3, chemin de données de 16 bits) et celle de base (degré 4, chemin de données de 18 bits). La seconde version correspond à celle que l'on aurait implantée sans l'aide de notre méthode. La table 1 donne les différentes caractéristiques des deux implantations pour un étage de Horner (logique et registres). L'optimisation permet d'obtenir un circuit 17% plus petit mais surtout d'utiliser une approximation de degré 3 plutôt que 4 et donc de gagner 38% en temps de calcul.

version	surface [slices]	période [ns]	nb cycles	durée du calcul [ns]
degré 3, $n'=16$	193	21.9	3	65.7
degré 4, $n'=18$	233	26.9	4	107.6

TAB. 1 – Résultats de synthèse pour 2^x sur $[0,1]$.

4.2. Racine carrée sur $[1,2]$

Pour ce deuxième exemple, nous cherchons à concevoir un opérateur très rapide pour évaluer $\sqrt{x}$ avec x dans $[1,2]$ et une précision d'au moins 8 bits au final (approximation et évaluation). Le polynôme minimax de degré 1 ne conduit qu'à 6.81 bits de précision, il faut au moins un polynôme de degré 2.

Le polynôme minimax de degré 2 pour $\sqrt{x}$ avec x dans $[1,2]$ est $0.4456804579 + 0.6262821240x - 0.7119874509x^2$. Il fournit une erreur d'approximation théorique de 0.0007638369 soit 10.35 bits corrects. La précision théorique supérieure à 10 bits permet de supposer que l'on devrait atteindre notre but si on trouve des coefficients représentables sans trop diminuer l'erreur d'approximation.

Toutefois, implanter directement ce polynôme, n'est pas une bonne idée du point de vue du format. Avec x dans $[1,2]$, il faut travailler un nombre de bits variable pour la partie entière. En effet, l'opération x^2 nécessite deux bits entiers alors que les autres seulement un. Pour éviter ceci, on utilise un changement de variable pour évaluer $\sqrt{1+x}$ avec x dans $[0,1]$. Le polynôme minimax correspondant est : $1.0007638368 + 0.4838846338x - 0.7119874509x^2$. On obtient la même erreur d'approximation de 10.35 bits corrects. Ceci est tout à fait normal car le changement de variable $x = 1 + x$ utilisé ne modifie pas la qualité du polynôme minimax.

A partir de ce polynôme, on pourrait procéder comme pour l'exemple 2^x, mais les coefficients p_0 et p_1 semblent très proches de puissances de 2 et on va essayer de l'utiliser. La première chose à faire est de remplacer p_0 par 1. Le polynôme $1 + 0.4838846338x - 0.7119874509x^2$ offre une précision d'approximation de 9.35 bits, ce qui nous semble satisfaisant.

Le coefficient p_1 semble proche de 0.5. Le polynôme $1 + 0.5x - 0.7119874509x^2$ offre une précision d'approximation de 6.09 bits seulement. p_1 ne peut donc pas être remplacé par 0.5. Toutefois nous allons essayer d'écrire p_1 avec peu de bits à 1 ou -1. Le coefficient p_1 est très proche de $(0.10000\overline{1})_2$. Le polynôme $1 + (0.10000\overline{1})_2x - 0.7119874509x^2$ offre une précision d'approximation de 9.45 bits et en plus le produit p_1x est remplacé par la soustraction $\frac{1}{2}x - \frac{1}{2^6}x$.

Nous procédons à une recherche d'une version avec peu de bits non-nuls de p_2 et nous trouvons $(0.0001001)_2$. Donc le produit p_2x^2 est remplacé par l'addition $\frac{1}{2^4}x^2 + \frac{1}{2^7}x^2$. Le polynôme $1 + (0.10000\overline{1})_2x + (0.0001001)_2x^2$ fournit une précision d'approximation de 9.49 bits. Il ne reste donc plus qu'une seule multiplication pour le calcul de x^2.

On va donc déterminer la précision finale intégrant l'erreur d'évaluation en utilisant GAPPA. En cherchant la taille n' du chemin de données on trouve 10 bits. Le programme à faire prouver par GAPPA est le suivant.

```
p0 = 1; p1 = 31/64; p2 = -9/128;
x = fixed<-10,dn>(Mx);
x2  fixed<-10,dn>= x * x;
p   fixed<-10,dn>= p2 * x2 + p1 * x + p0;
Mp = p2 * (Mx*Mx) + p1 * Mx + p0;
{ Mx in [0,1] /\ |Mp-Mf| in [0,0.0013829642] -> |p-Mf| in ? }
```

GAPPA retourne une erreur totale de 8.03 bits. Mais ce programme GAPPA correspond à l'utilisation de mutiplieurs pour effectuer les produits p_1x et p_2x^2. En pratique nous remplaçons ces multiplieurs par des additions/soustractions. Il faut donc donner à GAPPA une description exacte de ce qui est fait par notre architecture. La détermination de la taille minimale du chemin de données est faite en partant de $n = 8$ et en incrémentant n jusqu'à ce que la précision finale soit supérieure ou égale à 8 bits. La recherche donne $n = 13$.

```
p0  = 1;
p11 = 1/2;   p12 = -1/64;
p21 = -1/16; p22 = -1/128;
x = fixed<-8,dn>(Mx);
x2  fixed<-16,dn>= x * x;
p   fixed<-13,dn>= p21 * x2 + p22 * x2 + p11 * x + p12 * x + p0;
Mx2 = Mx * Mx;
Mp  = p21 * Mx2 + p22 * Mx2 + p11 * Mx + p12 * Mx + p0;
{ Mx in [0,1] /\ |Mp-Mf| in [0,0.0013829642] -> |p-Mf| in ? }
```

Dans ce cas, GAPPA retourne une précision de 8.07 bits avec une seule vraie multiplication pour x^2. L'architecture de l'opérateur est présentée en figure 3. Les cercles gris indiquent un décalage vers la droite du nombre de bits indiqué à l'intérieur du cercle (routage uniquement).

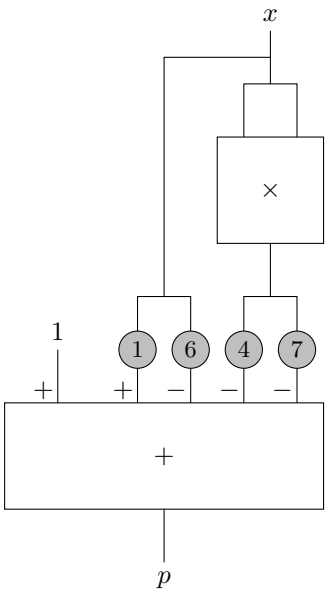


FIG. 3 – Architecture de l'opérateur optimisé pour $\sqrt{1+x}$ sur $[0,1]$.

Nous avons implanté la solution optimisée présentée en figure 3 et celle que l'on aurait implantée sans l'aide de la méthode (degré 2, étage de Horner avec chemin de données sur 11 bits). Les résultats de synthèse correspondants sont présentés dans la table 2. Ici aussi, les gains sont intéressants puisque l'on obtient une amélioration de 40% en surface et de 51% en temps de calcul.

version	surface [slices]	période [ns]	nb cycles	durée du calcul [ns]
degré 2 Horner	103	19.9	2	39.8
degré 2 optimisée	61	19.4	1	19.4

TAB. 2 – Résultats de synthèse pour $\sqrt{1+x}$ sur $[0,1]$.

5. Conclusion et perspectives

Nous avons proposé une méthode pour concevoir et optimiser des opérateurs arithmétiques matériels dédiés à l'évaluation de fonctions par approximation polynomiale. Notre méthode permet, à l'aide d'outils récents, de déterminer une solution avec :
– un degré d petit ;
– une taille de coefficients représentables n petite ;
– et une taille du chemin de données n' petite.
La méthode ne fournit pas des valeurs de d, n et n' optimales. L'optimum pour ce problème n'est pas connu actuellement même sur le plan théorique. Sur les exemples testés, la méthode permet d'obtenir des circuits plus petits et plus rapides que ceux que l'on pouvait concevoir avant. Toutefois, il reste beaucoup à faire pour les améliorer encore.
De plus, notre méthode permet, grâce à l'utilisation du logiciel GAPPA [7], d'obtenir des opérateurs valides numériquement dès la conception. En effet, la méthode permet de déterminer à la fois les erreurs d'approximation et les erreurs d'évaluation. Il n'est plus nécessaire de qualifier *a posteriori* le circuit avec de longues simulations ou tests. La contrainte de précision est vérifiée à la conception.

Dans l'avenir, nous pensons travailler dans plusieurs directions. Pour minimiser l'erreur d'approximation, nous poursuivons nos travaux présentés dans [2] et [1]. Pour minimiser l'erreur d'évaluation tout en évaluant rapidement le polynôme, il reste énormement de travail à faire. En utilisant, par exemple, d'autres schémas d'évaluation de polynômes comme celui proposé par Estrin pour certaines valeurs de *d*. Enfin, nous travaillons sur l'intégration de cette méthode dans des outils pour générer automatiquement des circuits.

Remerciements

Les auteurs tiennent à remercier chaleureusement Guillaume Melquiond pour son aide sur l'utilisation de son outil GAPPA.

Bibliographie

1. Brisebarre (N.), Muller (J.-M.) et Tisserand (A.). – Sparse-coefficient polynomial approximations for hardware implementations. *In : Proc. 38th Asilomar Conference on Signals, Systems and Computers*, pp. 532–535. – Pacific Grove, California, U.S.A., novembre 2004.
2. Brisebarre (N.), Muller (J.-M.) et Tisserand (A.). – Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, vol. 32, n° 2, juin 2006, pp. 236–256.
3. Chesneaux (J.-M.), Didier (L.-S.), Jézéquel (F.), Lamotte (J.-L.) et Rico (F.). – CADNA : Control of accuracy and debugging for numerical applications. – `http://www-anp.lip6.fr/cadna/`. LIP6–Univ. Pierre et Marie Curie.
4. Cordesses (L.). – Direct digital synthesis : A tool for periodic wave generation (part 1). *IEEE Signal Processing Magazine*, vol. 21, n° 4, juillet 2004, pp. 50–54.
5. Ercegovac (M. D.) et Lang (T.). – *Digital Arithmetic*. – Morgan Kaufmann, 2003.
6. Goubault (E.), Martel (M.) et Putot (S.). – FLUCTUAT : Static analysis for numerical precision. – `http://www-list.cea.fr/labos/fr/LSL/fluctuat/`. CEA-LIST.
7. Melquiond (G.). – GAPPA : génération automatique de preuves de propriétés arithmétiques. – `http://lipforge.ens-lyon.fr/www/gappa/`. Arénaire, LIP, CNRS-ENSL-INRIA-UCBL.
8. Ménard (D.) et Sentieys (O.). – Automatic evaluation of the accuracy of fixed-point algorithms. *In : Proc. Design, Automation and Test in Europe (DATE)*, éd. par Kloos (C. D.) et da Franca (J.), pp. 529–537. – mars 2002.
9. Muller (J.-M.). – *Elementary Functions : Algorithms and Implementation*. – Birkhäuser, 2006, 2nd édition.
10. Remes (E.). – Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, vol. 198, 1934, pp. 2063–2065.

Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales

Romain Michard[*] — **Arnaud Tisserand**[**] — **Nicolas Veyrat-Charvillon**[*]

[*] *Arénaire, LIP, CNRS–ENS Lyon–INRIA–UCB Lyon*
46 allée d'Italie. F-69364 Lyon
{romain.michard,nicolas.veyrat-charvillon}@ens-lyon.fr

[**] *Arith, LIRMM, CNRS–Univ. Montpellier 2*
161 rue Ada. F-34392 Montpellier
arnaud.tisserand@lirmm.fr

RÉSUMÉ. *Cet article présente une méthode pour l'optimisation d'opérateurs arithmétiques matériels dédiés à l'évaluation de fonctions en utilisant des polynômes d'approximation. La méthode, basée sur des outils récents, réduit la taille des coefficients des polynômes et des valeurs intermédiaires tout en bornant l'erreur totale (approximation et évaluation). Elle conduit à des opérateurs petits et rapides tout en garantissant une bonne qualité numérique. La méthode est illustrée sur quelques exemples sur des circuits FPGA.*

ABSTRACT. *This article presents a method for the optimisation of hardware arithmetic operators dedicated to function evaluation using polynomial approximations. Using recent tools, the method reduces the size of the polynomial coefficients and the intermediate values while keeping the total error bounded (approximation and evaluation). It leads to small and fast operators with a good numerical quality. The method is illustrated on several examples implemented on FPGA circuits.*

MOTS-CLÉS : *arithmétique des ordinateurs, circuit intégré numérique, approximation polynomiale, évaluation de polynôme.*

KEYWORDS: *computer arithmetic, digital integrated circuit, polynomial approximation, polynomial evaluation*

1. Introduction

Les systèmes sur puces embarquent de plus en plus de calculs complexes. En plus des additionneurs et multiplieurs rapides, le support matériel d'opérations évoluées est nécessaire pour certaines applications spécifiques. Des applications en traitement du signal et des images effectuent des divisions, inverses, racines carrées ou racines carrées inverses. D'autres applications utilisent des sinus, cosinus, exponentielles ou logarithmes comme en génération numérique de signaux (Cordesses, 2004).

Différents types de méthodes sont proposés dans la littérature pour l'évaluation de fonctions. On trouve dans (Ercegovac *et al.*, 2003) les bases d'arithmétique des ordinateurs et en particulier les opérations comme la division, l'inverse, la racine carrée et la racine carrée inverse. Dans (Muller, 2006), les principales méthodes d'évaluation des fonctions élémentaires ($\sin$, $\cos$, $\exp$, $\log\ldots$) sont présentées.

Parmi toutes les méthodes possibles, les approximations polynomiales sont souvent utilisées. Elles permettent d'évaluer efficacement la plupart des fonctions que l'on trouve dans les applications embarquées. L'évaluation pratique d'un polynôme s'effectue en utilisant des additions et des multiplications. La surface des multiplieurs est souvent très importante dans le circuit. Ceci limite le degré du polynôme d'approximation ou alors il faut recourir à des opérateurs effectuant plusieurs itérations pour parvenir au résultat final.

En section 2, nous présentons les notations, l'état de l'art du domaine et en particulier un outil récent permettant de borner finement les erreurs: GAPPA (Melquiond, 2005–2007). Dans ce travail, nous montrons comment obtenir des opérateurs spécifiques pour l'évaluation de fonctions par des approximations polynomiales optimisées. La méthode, proposée en section 3, intervient à deux niveaux pour réaliser cette optimisation. À partir du meilleur polynôme d'approximation théorique, nous déterminons des coefficients propices à une implantation matérielle efficace. Ensuite, nous montrons comment réduire la taille des valeurs intermédiaires en utilisant GAPPA. Notre méthode permet de construire des approximations polynomiales à la fois efficaces et garanties numériquement à la conception. Dans la plupart des méthodes précédentes, seule l'erreur d'approximation était prise en compte pour l'optimisation. Ici, l'erreur totale, approximation et évaluation, est optimisée. Enfin, nous illustrons en section 4 l'impact de nos optimisations pour quelques exemples de fonctions sur des circuits FPGA (*field programmable gate array*).

2. Notations et état de l'art

2.1. *Notations*

La représentation en virgule fixe est la plus courante pour les applications de traitement du signal et des images. Dans la suite, nous supposons toutes les valeurs représentées en virgule fixe. Une extension de ce travail à la virgule flottante est assez simple en pratique du fait des domaines de départ et d'arrivée des fonctions cibles. La

fonction à évaluer est f pour l'argument x du domaine $[a, b]$. L'argument x est dans un format sur n bits tandis que le résultat $f(x)$ est sur m bits, souvent on a $n \approx m$. Le polynôme d'approximation p utilisé pour approcher f est de degré d, ses coefficients sont notés: p_0, p_1,..., p_d. On a donc $p(x) = \sum_{i=0}^{d} p_i x^i$. La représentation binaire des valeurs est notée $()_2$, par exemple $(11.01)_2$ représente la valeur décimale 3.25. La notation binaire signée *borrow-save* est utilisée pour les coefficients (notation en base 2 avec les chiffres $\{-1, 0, 1\}$, cf. (Ercegovac *et al.*, 2003)) avec le chiffre -1 noté $\overline{1}$.

2.2. *Erreurs*

L'argument x est considéré comme exact. L'*erreur d'approximation* ϵ_{app} est la distance entre la fonction mathématique f et le polynôme d'approximation p utilisé pour approcher f. Pour déterminer cette erreur, nous utilisons la *norme infinie* définie ci-dessous et estimée numériquement par MAPLE (fonction `infnorm`).

$$\epsilon_{\text{app}} = ||f - p||_\infty = \max_{a \leq x \leq b} |f(x) - p(x)|.$$

Du fait de la précision finie des calculs, des erreurs d'arrondi se produisent au cours de l'évaluation du polynôme p en machine. L'*erreur d'évaluation* ϵ_{eval} est due à l'accumulation des erreurs d'arrondi. Même si l'erreur d'évaluation commise lors d'une seule opération est toute petite (une fraction du poids du dernier bit), l'accumulation de ces erreurs peut devenir catastrophique lors de longues séries de calculs si rien n'est prévu pour en limiter les effets.

L'erreur d'évaluation est très difficile à borner finement (Ménard *et al.*, 2002). Le problème est lié au fait que cette erreur dépend de la valeur des opérandes. Elle est donc difficile à connaître *a priori*. Souvent, on considère le pire cas d'arrondi pour chaque opération. C'est à dire la perte de précision correspondant à la moitié du bit de poids le plus faible ($1/2$ LSB (*least significant bit*) en arrondi au plus près). La perte de précision est au maximum de 1 LSB en arrondi dirigé. En virgule fixe, ce qui sera notre cas, on arrive à $1/2$ LSB de précision d'arrondi en utilisant des techniques de biais. Pour l'évaluation d'un polynôme de degré d en utilisant le schéma de Horner (d additions et d multiplications), cette borne pire cas correspond à une perte de d bits de précision. Nous allons voir dans la suite que nous pouvons faire des approximations bien moins pessimistes en utilisant l'outil GAPPA développé par G. Melquiond (cf. section 2.5).

Il est possible de mesurer *a posteriori* l'erreur totale commise pour une valeur particulière de l'entrée x en utilisant $f(x) - \texttt{output}(p(x))$ où $\texttt{output}(p(x))$ est la sortie réelle (physique ou simulée par un simulateur au niveau bit) de l'opérateur. Cette technique est souvent utilisée pour qualifier *a posteriori* la précision d'opérateurs par des simulations exhaustives ou pour les entrées menant aux plus grandes erreurs[1]. D'une manière générale, cette technique de qualification *a posteriori* des opérateurs

1. Ces entrées particulières étant très difficiles à déterminer dans le cas général.

de calcul est coûteuse en temps de simulation et de conception si il faut modifier certains paramètres des opérateurs.

Dans la suite, nous exprimons les erreurs soit directement soit en terme de précision équivalente. Cette dernière est le nombre de bits n_c justes ou corrects équivalents à une certaine erreur ϵ avec $n_c = -\log_2 |\epsilon|$ pour une représentation fractionnaire du type $x_0.x_1x_2 \dots x_n$. Ceci signifie aussi que lorsque l'on a n_c bits justes, l'erreur absolue commise est alors plus petite que 2^{-n_c}. Par exemple, une erreur de 2.3×10^{-3} est équivalente à une précision de 8.7 bits corrects. Calculer avec 12 bits de précision signifie avoir une erreur d'au plus $2^{-12} = 0.00024414$.

2.3. *Polynôme minimax*

Il existe différents types de polynômes d'approximation (Muller, 2006). Le polynôme *minimax* est, en quelque sorte, le meilleur polynôme d'approximation pour notre problème. Le polynôme minimax de degré d pour approcher f sur $[a, b]$ est le polynôme p^* qui satisfait:

$$||f - p^*||_\infty = \min_{p \in \mathcal{P}_d} ||f - p||_\infty,$$

où $\mathcal{P}_d$ est l'ensemble des polynômes à coefficients réels de degré au plus d. Ce polynôme peut être déterminé numériquement grâce à l'algorithme de Remes (Remes, 1934) (implanté dans la fonction MAPLE `minimax`). Le polynôme minimax est le meilleur car parmi tous les polynômes d'approximation possibles de f sur $[a, b]$, il est celui qui a la plus petite des erreurs maximales ϵ_{app} sur tout l'intervalle. On note ϵ^*_{app} l'erreur d'approximation du polynôme minimax avec $\epsilon^*_{\text{app}} = ||f - p^*||_\infty$.

Mais le polynôme minimax est un polynôme « théorique » car ses coefficients sont dans $\mathbb{R}$ et on suppose son évaluation exacte (c'est à dire faite avec une précision infinie). Pour pouvoir évaluer ce polynôme en pratique, il faut faire deux niveaux d'approximation supplémentaires dont les erreurs peuvent se cumuler avec l'erreur d'approximation théorique ϵ^*_{app}:

– les coefficients de p^* devront être écrits en précision finie dans le format supporté par le circuit;

– les calculs nécessaires à son évaluation seront effectués en précision finie.

Pour le moment, il n'existe pas de méthode théorique permettant d'obtenir le meilleur polynôme d'approximation en pratique (coefficients *et* évaluation dans la précision du format cible). Mais nous allons montrer dans la suite que d'importantes optimisations sont déjà possibles avec une méthode assez simple et des outils facilement accessibles. La figure 1 illustre la problèmatique de l'implantation pratique de polynômes d'approximation en matériel.

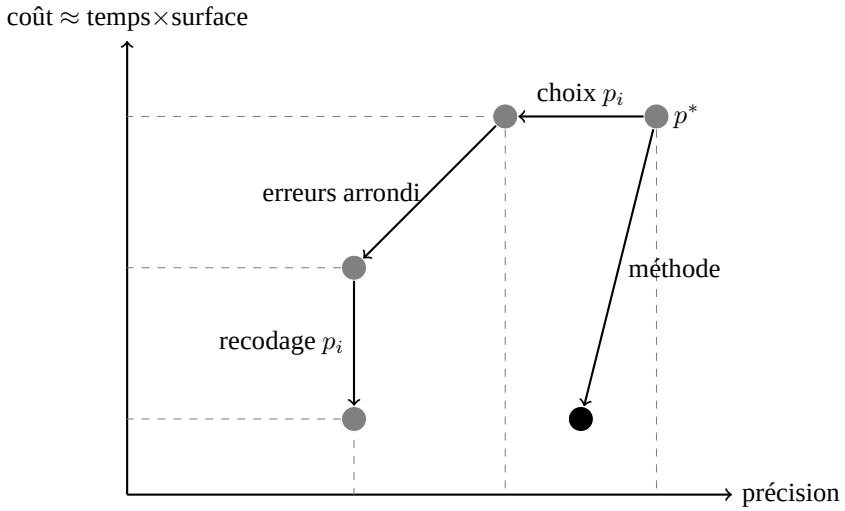


Figure 1. *Résumé du problème d'implantation de polynômes d'approximation en matériel.*

2.4. *Vers des polynômes d'approximation avec des coefficients représentables*

L'un des problèmes pour l'implantation pratique d'approximations polynomiales est le fait que les coefficients du polynôme minimax ne sont pas représentables dans le format cible en précision finie. Ceci est vrai aussi pour les autres types d'approximations polynomiales(Chebyshev, Legendre…), cf. (Muller, 2006).

Par exemple, prenons la fonction e^x sur l'intervalle $[1/2, 1]$ avec un polynôme de degré 2 et des coefficients sur 10 bits dont 1 en partie entière, soit $x = (x_0.x_1x_2 \ldots x_9)_2$. Le polynôme minimax trouvé est[2] $1.116018832 + 0.535470996x + 1.065407154x^2$, il conduit à une précision d'approximation de 9.4 bits environ. C'est la meilleure précision atteignable avec un polynôme de degré 2 sur $[1/2, 1]$. En arrondissant au plus près les 3 coefficients de ce polynôme dans le format cible, on a alors $\frac{571}{512} + \frac{137}{256}x + \frac{545}{512}x^2$ et une précision de l'approximation de 8.1 bits. Après avoir testé tous les modes d'arrondi possibles pour chacun des coefficients dans le format cible, on trouve que le meilleur polynôme est $\frac{571}{512} + \frac{275}{512}x + \frac{545}{512}x^2$ pour une précision de l'approximation de 9.3 bits, soit un gain de 1.2 bit de précision.

Dans les dernières années, différents travaux de recherche ont été menés pour obtenir des « bons » polynômes d'approximation dont les coefficients soient exactement représentables dans le format cible.

2. en arrondissant les coefficients théoriques sur 10 chiffres décimaux soit envion 32 bits.

On trouve dans (Brisebarre *et al.*, 2006) une méthode basée sur une formulation en un problème de programmation linéaire en nombres entiers. Les coefficients des polynômes sont représentés par des rationnels dont le dénominateur est une puissance de 2 (2^n pour un format fractionnaire sur n bits). Un polytope est construit à partir des contraintes (les tailles) sur tous les coefficients du polynôme. Chaque point du polytope est alors un polynôme dont les coefficients s'écrivent avec les tailles souhaitées. Mais seuls certains de ces polynômes sont des « bons » polynômes d'approximation de f. La liste des polynômes résultats est filtrée en deux phases. L'erreur d'approximation est mesurée en un certain nombre de points d'échantillonage pour chaque point (polynôme) possible du polytope. Le parcours du polytope donne une liste de polynômes pour lesquels la distance entre la fonction et le polynôme testé est plus petite qu'un seuil pour chacun des points d'echantillonage. Cette première liste de polynômes est ensuite traitée pour déterminer numériquement les normes infinies entre chacun des polynômes de la liste et la fonction. Cette méthode, séduisante sur le plan théorique, ne fonctionne pas encore très bien en pratique. Le temps de calcul et le volume de mémoire nécessaires sont trop importants pour le moment.

L'article présenté dans (Brisebarre *et al.*, 2004) utilise (Brisebarre *et al.*, 2006) pour produire des approximations polynomiales avec des coefficients creux (avec beaucoup de 0) pour des petits degrés (3 ou 4). Toutefois, du fait des énormes besoins en temps de calcul et en taille mémoire de (Brisebarre *et al.*, 2006), les approximations trouvées sont limitées à une douzaine de bits de précision. De plus, seule l'erreur d'approximation est prise en compte.

2.5. *Outils pour le calcul de bornes d'erreur globale*

L'étude de méthodes pour borner finement les erreurs de calcul est un domaine de recherche très actif depuis quelques années. Dans le domaine du traitement du signal, des méthodes assimilant les erreurs d'arrondi à du bruit sont utilisées avec succès (Ménard *et al.*, 2002). Différents outils ont été développés pour le calcul scientifique en virgule flottante. Par exemple, FLUCTUAT (Goubault *et al.*, n.d.) est un analyseur statique de code qui permet de détecter certaines pertes de précision sur des programmes flottants. Le logiciel CADNA (Chesneaux *et al.*, n.d.) implante une méthode stochastique pour l'analyse de la précision moyenne des programmes flottants (une version en virgule fixe est actuellement étudiée).

Le logiciel GAPPA (Melquiond, 2005–2007) permet d'évaluer et de prouver des propriétés mathématiques sur des programmes numériques. La caractéristique intéressante de GAPPA dans notre problème est sa capacité à borner finement les erreurs de calcul ou à montrer que des bornes sont en dessous d'un certain seuil.

Voici un exemple simple des possibilités de GAPPA sur la fonction e^x sur $[1/2, 1]$. On suppose tous les calculs effectués en virgule fixe sur 10 bits dont 1 en partie entière. Le polynôme d'approximation utilisé est $p(x) = \frac{571}{512} + \frac{275}{512}x + \frac{545}{512}x^2$, son évaluation se décrit ainsi en GAPPA:

```
1 p0 = 571/512; p1 = 275/512; p2 = 545/512;
2 x = fixed<-9,dn>(Mx);
3 x2 fixed<-9,dn>= x * x;
4 p fixed<-9,dn>= p2 * x2 + p1 * x + p0;
5 Mp = p2 * (Mx*Mx) + p1 * Mx + p0;
6 { Mx in [0.5,1] /\ |Mp-Mf| in [0,0.001385]
7   -> |p-Mf| in ? }
```

La ligne 1 spécifie les coefficients du polynôme (choisis représentables dans le format considéré). Par convention dans la suite, les noms de variables qui commencent par un `M` majuscule représentent les valeurs mathématiques (en précision infinie), et toutes les valeurs en miniscules représentent des variables du programme (des entrées/sorties ou des registres intermédiaires). La ligne 2 indique que `x` est la version circuit de l'argument mathématique `Mx` (en précision infinie). La construction `fixed<-9,dn>` indique que l'on travaille en virgule fixe avec le LSB de poids 2^{-9} et l'arrondi vers le bas `dn` (troncature). La ligne 3 indique que la variable `x2` est la version calculée dans le circuit de x^2. La ligne 4 décrit comment le polynôme est évalué en pratique dans le circuit tandis que la ligne 5 décrit son évaluation théorique (en précision infinie car sans l'opérateur d'arrondi `fixed<.,.>`). Enfin, les lignes 6 et 7 indiquent la propriété cherchée (entre accolades). Les hypothèses sont en partie gauche du signe `->`, elles indiquent que la valeur mathématique de x est dans $[1/2, 1]$ et que l'erreur d'approximation entre le polynôme d'appromixation `Mp` (sans erreur d'arrondi) et la fonction mathématique `Mf` (la fonction cible théorique sans aucune erreur) est inférieure ou égale à 0.001385 (valeur fournie par MAPLE). La partie droite du signe `->` indique que l'on demande à GAPPA de nous dire dans quel intervalle (`in ?`) est la distance entre la valeur évaluée dans le circuit du polynôme `p` et la fonction mathématique, en incluant erreurs d'approximation et d'évaluation (arrondis).

Le résultat retourné par GAPPA (version supérieure ou égale à `0.6.1`) pour ce calcul est:

```
Results for Mx in [0.5, 1] and |Mp - F| in [0, 0.001385]:
|p - F| in [0, 232010635959353905b-64 {0.0125773, 2^(-6.31303)}]
```

Après avoir répété les hypothèses, GAPPA indique qu'il a trouvé une borne pour $|p - f|$ et qu'il y ait au moins 6.31 bits corrects.

En modifiant les lignes 3 à 5 par les lignes suivantes, on cherche l'erreur totale commise en utilisant le schéma de Horner, GAPPA indique alors une précision globale de 6.57 bits.

```
3 y1 fixed<-9,dn>= p2 * x + p1;
4 p  fixed<-9,dn>= y1 * x + p0;
5 Mp = (p2 * Mx + p1) * Mx + p0;
```

3. Méthode d'optimisation proposée

Dans un premier temps, nous présentons rapidement la méthode, ensuite nous donnerons des détails sur chacune des différentes étapes et les informations sur les outils nécessaires.

Les données nécessaires en entrée de la méthode sont:

– f la fonction à évaluer;

– $[a, b]$ le domaine de l'argument x;

– le format de l'argument x (nombre de bits n_x);

– μ l'erreur totale maximale cible (donnée en erreur absolue).

Les paramètres déterminés par la méthode sont:

– d le degré du polynôme à utiliser;

– p_0, p_1, p_2,..., p_d les valeurs des coefficients du polynôme représentables dans le circuit;

– n la taille utilisée pour représenter les coefficients;

– n' la taille utilisée pour effectuer les calculs[3].

Notre méthode se résume aux 3 étapes ci-dessous avec des retours possibles à une étape antérieure (rebouclage) dans certains cas:

Étape 1: calcul du *polynôme minimax*.
On cherche p^* de degré d le plus petit possible tel que $\epsilon^*_{\text{app}} < \mu$ avec $\epsilon^*_{\text{app}} = ||f - p^*||_\infty$. Cette première phase donne l'erreur d'approximation ϵ^*_{app} minimale atteignable en supposant tous les calculs faits et coefficients représentés en précision infinie.

Étape 2: détermination des *coefficients* du polynôme à implanter et de leur *taille*.
On cherche ici à la fois les p_i et leur taille minimale n telle que l'erreur d'approximation ϵ_{app} du polynôme p utilisé ($p(x) = \sum_{i=0}^{d} p_i x^i$) soit strictement inférieure à μ.

Étape 3: détermination de la *taille du chemin de données*.
On cherche n' la taille minimale du chemin de données de l'étage de Horner pour effectuer les calculs. Cette dernière phase donne l'erreur d'évaluation ϵ_{eval} qui intègre les erreurs d'approximation et les erreurs d'arrondi. La valeur de n' trouvée garantit que $\epsilon_{\text{eval}} < \mu$.

Dans certains cas, il n'y a pas de solution à une étape. Il faut alors *reboucler* à l'étape précédente pour essayer d'autres solutions.

Différents types de rebouclages sont possibles. Par exemple, en fin de deuxième étape, on peut ne pas trouver des coefficients représentables pour garantir $\epsilon_{\text{app}} < \mu$.

3. Nous verrons que dans certains cas, prendre n et n' légèrement différents peut aider à limiter la taille du circuit.

Ceci se produit lorsque la « marge » entre ϵ^*_{app} et μ était trop faible à la première étape. Il faut donc retourner à la première étape pour essayer un polynôme de degré plus grand $d \leftarrow d + 1$.

Un autre type classique de rebouclage intervient à la fin de la dernière étape. Si la taille n' trouvée est jugée trop grande devant n, il peut être intéressant de revenir à la deuxième étape pour essayer un n plus grand. Ceci permet d'avoir ϵ_{app} plus petit et donc plus de marge pour les erreurs d'arrondi.

La décision d'application de ces rebouclages est purement heuristique. Nous allons l'illustrer sur les exemples en section 4.

Dans les descriptions données jusqu'ici, nous utilisons des contraintes du style $\epsilon_{\text{app}} < \mu$ et pas $\epsilon_{\text{app}} \leq \mu$. En pratique, il faut de la marge entre ϵ^*_{app} et μ puis entre ϵ_{app} et μ. Le caractère stricte des contraintes pour les étapes 1 et 2 est donc nécessaire. Pour la dernière étape, peut être pouvons-nous trouver une fonction (probablement triviale et donc peu intéressante) telle que l'on ait $\epsilon_{\text{eval}} = \mu$ à la fin. Cela nous semble très improbable. De plus, l'utilisateur saurait traiter convenablement ce cas.

3.1. *Calcul du polynôme minimax*

Dans cette étape on utilise la fonction `minimax` de MAPLE. On commence avec $d = 1$, et on incrémente d jusqu'à ce que le polynôme minimax p^* trouvé soit tel que $\epsilon^*_{\text{app}} < \mu$.

Voici un exemple pour $f = \log_2(x)$ avec x dans $[1, 2]$:

```
1 > minimax(log[2](x),x=1..2,[1,0],1,'err'); -log[2](err);
2     -.95700010+1.0000000*x
3     4.5371245
4 > minimax(log[2](x),x=1..2,[2,0],1,'err'); -log[2](err);
5     -1.6749034+(2.0246817-.34484766*x)*x
6     7.6597968
7 > minimax(log[2](x),x=1..2,[3,0],1,'err'); -log[2](err);
8     -2.1536207+(3.0478841+(-1.0518750+.15824870*x)*x)*x
9     10.616152
```

Les lignes qui commencent par un signe supérieur (« prompt » MAPLE) sont les commandes entrées par l'utilisateur. Les résultats retournés par MAPLE sont en italique. La ligne 1 signifie que l'on cherche un polynôme de degré 1 et que l'erreur d'approximation trouvée sera placée dans la variable `err`. Les lignes 2 et 3 représentent respectivement le polynôme trouvé et sa précision (en nombre de bits corrects).

Cette étape fournit trois éléments nécessaires pour la suite:

– d le degré du polynôme;

– $p^*(x) = \sum_{i=0}^{d} p_i^* x^i$ le polynôme minimax (théorique);

– ϵ^*_{app} l'erreur *minimale* atteignable en utilisant p^* pour approcher f (en précision infinie).

Les deux autres étapes vont dégrader la qualité de l'approximation (i.e. fournir des erreurs plus grandes que ϵ^*_{app}). Il faut donc laisser un peu de marge entre ϵ^*_{app} et μ. Nous reviendrons sur cette marge.

Nous verrons dans l'exemple 4.2, que certains changements de variables permettent d'obtenir des coefficients d'ordres de grandeur voisins. Ceci limite les recadrages en virgule fixe. Si on évalue $f(x)$ sur $[a, b]$ avec $a \neq 0$, il peut être intéressant de considérer $f(x + a)$ sur $[0, b - a]$.

Toutes les fonctions ne sont pas « approchables facilement » avec des polynômes. Nous renvoyons le lecteur aux ouvrages de références sur l'évaluation de fonctions comme (Muller, 2006, chap. 3) pour les fonctions élémentaires. En pratique, les fonctions usuelles s'approchent bien par des polynômes.

3.2. *Détermination des coefficients du polynôme et de leur taille*

Une fois p^* déterminé, il faut trouver des coefficients représentables en précision finie. On cherche n le nombre de bits du format virgule fixe des p_i tel que n soit le plus petit possible mais avec $\epsilon_{\text{app}} < \mu$.

Nous avons vu en 2.4 que le choix des coefficients est important. En arrondissant simplement les coefficients du polynôme minimax sur n bits, il est peu probable de trouver un bon polynôme d'approximation. L'exemple présenté en 2.4 montre que tester l'ensemble des combinaisons des arrondis des coefficients de p^* permet de trouver un bon polynôme. C'est ce que nous proposons de faire systématiquement dans cette deuxième phase.

Chaque coefficient p^*_i du polynôme peut être arrondi soit vers le haut $p_i = \triangle(p^*_i)$, soit vers le bas $p_i = \triangledown(p^*_i)$. Il y a 2 choix possibles par coefficient et donc 2^{d+1} combinaisons à tester au total comme illustré en figure 2. Pour chaque polynôme p, il faut déterminer $\epsilon_{\text{app}} = ||f - p||_\infty$ (fonction `infnorm` de MAPLE).

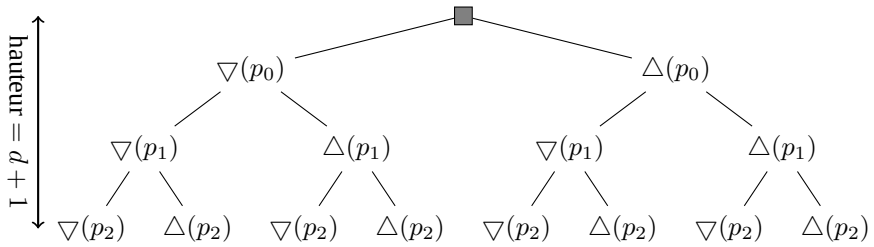


Figure 2. *Différentes combinaisons d'arrondi pour p^* de degré $d = 2$.*

Dans nos applications, d est faible ($d \leq 6$). Il y a donc au mieux quelques centaines de polynômes à tester. En pratique, chaque calcul de ϵ_{app} dure quelques fractions de seconde sur un ordinateur standard.

Nous avons fait un programme MAPLE qui teste les 2^{d+1} combinaisons d'arrondi des p_i^*. Le programme retourne la liste des polynômes candidats pour la troisième étape, c'est à dire ceux pour qui l'erreur d'approximation ϵ_{app} est minimale. On commence par $n = \lceil -\log_2 |\mu| \rceil$ (le nombre de bits correspondant à l'erreur μ), puis on test tous les arrondis des $d+1$ coefficients sur n bits. On recommence en incrémentant n jusqu'à ce que $\epsilon_{\text{app}} < \mu$.

A cette étape, on peut aussi souhaiter modifier plus en « profondeur » certains coefficients. Par exemple, si un coefficient retourné est $0.5002441406 = (0.100000000001)_2$ et que l'on travaille sur un peu plus d'une douzaine de bits fractionnaires, il est peut être intéressant de fixer ce coefficient à 0.5 pour éliminer une opération. La multiplication par une puissance de 2 se réduit à un décalage. Nous verrons un exemple de ce genre de modification dans l'exemple 4.2. Nous ne savons pas encore formaliser simplement ce genre de traitement, mais cela constitue un de nos axes de recherche à travers l'amélioration de la méthode présentée dans (Brisebarre *et al.*, 2006).

3.3. *Détermination de la taille du chemin de données*

Le calcul du polynôme $p(x)$ peut s'effectuer en utilisant différents schémas d'évaluation (Muller, 2006). Dans la suite, nous utiliserons uniquement les schémas direct et de Horner:

$$p(x) = \begin{cases} p_0 + p_1 x + p_2 x^2 + \cdots + p_d x^d & \text{direct } d \text{ add.}, d + \lceil \log_2 d \rceil \text{ mul.;} \\ p_0 + x(p_1 + x(p_2 + x(\cdots + x p_d) \cdots)) & \text{Horner } d \text{ add.}, d \text{ mul.} \end{cases}$$

Le schéma de Horner est souvent préféré au schéma direct car il nécessite moins d'opérations et donne souvent une erreur d'évaluation ϵ_{eval} plus faible (Muller, 2006). Toutefois, dans certains cas particuliers avec des coefficients très creux, le schéma direct peut s'avérer intéressant (cf. exemple en 4.2).

La dernière étape permet de spécifier la taille du chemin de données pour l'évaluation suivant le schéma de Horner. L'étage de Horner permet d'évaluer $u \times v + z$. La taille du chemin de données de cet étage est n'. On commence avec $n' = n$ et on augmente n' tant que l'encadrement de l'erreur d'évaluation ϵ_{eval} par GAPPA avec n' bits pour le chemin de données ne donne pas $\epsilon_{\text{eval}} < \mu$. Pour le moment, le codage en GAPPA se fait à la main. Mais le schéma de Horner ou le schéma direct étant les mêmes aux coefficients près, nous avons les programmes GAPPA types pour lesquels il suffit de préciser les valeurs des coefficients et du degré. La boucle sur n' s'effectue en quelques minutes tout au plus sur un ordinateur standard.

La différence entre n' et n est appelée le nombre de bits de garde. Conserver n plus petit que n' permet, par exemple, de limiter la taille mémoire nécessaire pour stocker les coefficients.

Si la taille du chemin de données est un peu supérieure à n (1 à 3 bits), on pourrait revenir à l'étape 2 pour essayer un n plus grand. Notre expérience montre que le n' final change rarement. Par contre si la valeur de n' est bien plus grande que n, alors il peut être intéressant de reboucler à l'étape 2 avec un n plus grand. Ici encore, nous devons encore travailler pour formaliser ce genre de test.

3.4. *Résumé de la méthode*

La figure 3 résume graphiquement le fonctionnement de la méthode.

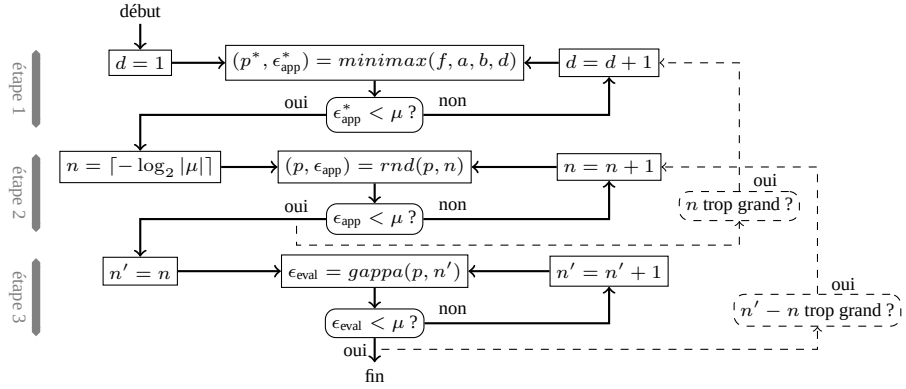


Figure 3. *Résumé de la méthode (les rebouclages en pointillés sont facultatifs).*

4. Exemples d'applications sur FPGA

Les implantations réalisées ci-dessous ont été faites pour des FPGA de la famille Virtex de Xilinx (XCV200-5) avec les outils ISE8.1i de Xilinx. La synthèse et le placement/routage utilisent une optimisation en surface avec effort élevé. Les résultats indiquent toutes les ressources nécessaires pour chaque opérateur (cellules logiques et registres fonctionnels). Nous avons utilisé des circuits FPGA pour ces premières implantations du fait de leur simplicité d'utilisation. Dans l'avenir nous souhaitons optimiser et implanter nos opérateurs pour des circuits intégrés standards (ASIC). La méthode présentée dans cet article est indépendante de la cible d'implantation.

4.1. *Fonction* 2^x *sur* $[0,1]$

On cherche un opérateur pour évaluer la fonction 2^x, avec x dans $[0,1]$ et une précision globale de 12 bits. La première étape est de trouver un polynôme d'approximation théorique en utilisant MAPLE. On cherche ici à minimiser le degré d du polynôme utilisé. Voici la précision d'approximation ϵ_{app} (en bits corrects) des polynômes minimax de 2^x pour d variant de 1 à 5:

d	1	2	3	4	5
ϵ_{app}	4.53	8.65	13.18	18.04	23.15

Bien évidemment, les polynômes de degré 1 et 2 ne sont pas suffisament précis pour notre but. La solution avec le polynôme de degré 3 conduit à précision estimée dans le pire cas à 10 bits environ ($13.18 - d$, car on borne à d bits la perte de précision pour un schéma de Horner de degré d) ce qui semble trop faible. De plus, les 13.18 bits corrects correspondent à l'erreur d'approximation avec le polynôme minimax avec des coefficients réels (non représentables dans le format cible).

Sans outil pour aider le concepteur, il semble donc que l'on serait obligé de choisir la solution de degré 4 ou 5. Même la solution avec un polynôme de degré 4 peut conduire à une précision trop faible. Certes sa précision théorique est au moins de $18.04 - 4 = 14.04$ bits corrects mais seulement pour des coefficients en précision infinie. Le polynôme minimax de degré 4 est (les coefficients sont affichés avec 10 chiffres décimaux dans le papier, mais MAPLE est capable de les calculer avec plus de précision): $1.0000037045 + 0.6929661227x + 0.2416384458x^2 + 0.0516903583x^3 + 0.0136976645x^4$.

Pour être certain que la solution de degré 4 peut être employée, il faut trouver un format des coefficients pour lequel le polynôme minimax avec ses coefficients arrondis dans ce format ait une précision supérieure ou égale à $12+4 = 16$ bits. Pour un format donné, on teste tous les modes d'arrondi possibles pour les coefficients du polynôme minimax dans le format. On trouve des polynômes acceptables à partir de 14 bits fractionnaires et 1 entier. Pour montrer que le choix de coefficients représentables est important, nous présentons ci-dessous chacune des combinaisons possibles des modes d'arrondi des coefficients et la précision d'approximation du polynôme dont les coefficients s'écrivent exactement dans le format cible. Seuls deux polynômes ont la précision souhaitée (valeurs en gras).

$(\bigtriangledown,\bigtriangledown,\bigtriangledown,\bigtriangledown,\bigtriangledown)$	12.00	$(\bigtriangledown,\bigtriangledown,\bigtriangledown,\bigtriangledown,\bigtriangleup)$	13.00
$(\bigtriangledown,\bigtriangledown,\bigtriangledown,\bigtriangleup,\bigtriangledown)$	13.00	$(\bigtriangledown,\bigtriangledown,\bigtriangledown,\bigtriangleup,\bigtriangleup)$	14.03
$(\bigtriangledown,\bigtriangledown,\bigtriangleup,\bigtriangledown,\bigtriangledown)$	13.00	$(\bigtriangledown,\bigtriangledown,\bigtriangleup,\bigtriangledown,\bigtriangleup)$	14.55
$(\bigtriangledown,\bigtriangledown,\bigtriangleup,\bigtriangleup,\bigtriangledown)$	14.99	$(\bigtriangledown,\bigtriangledown,\bigtriangleup,\bigtriangleup,\bigtriangleup)$	13.00
$(\bigtriangledown,\bigtriangleup,\bigtriangledown,\bigtriangledown,\bigtriangledown)$	13.00	$(\bigtriangledown,\bigtriangleup,\bigtriangledown,\bigtriangledown,\bigtriangleup)$	**16.13**
$(\bigtriangledown,\bigtriangleup,\bigtriangledown,\bigtriangleup,\bigtriangledown)$	**17.12**	$(\bigtriangledown,\bigtriangleup,\bigtriangledown,\bigtriangleup,\bigtriangleup)$	13.00
$(\bigtriangledown,\bigtriangleup,\bigtriangleup,\bigtriangledown,\bigtriangledown)$	15.71	$(\bigtriangledown,\bigtriangleup,\bigtriangleup,\bigtriangledown,\bigtriangleup)$	13.00
$(\bigtriangledown,\bigtriangleup,\bigtriangleup,\bigtriangleup,\bigtriangledown)$	13.00	$(\bigtriangledown,\bigtriangleup,\bigtriangleup,\bigtriangleup,\bigtriangleup)$	12.00
$(\bigtriangleup,\bigtriangledown,\bigtriangledown,\bigtriangledown,\bigtriangledown)$	13.00	$(\bigtriangleup,\bigtriangledown,\bigtriangledown,\bigtriangledown,\bigtriangleup)$	13.00
$(\bigtriangleup,\bigtriangledown,\bigtriangledown,\bigtriangleup,\bigtriangledown)$	13.00	$(\bigtriangleup,\bigtriangledown,\bigtriangledown,\bigtriangleup,\bigtriangleup)$	13.00
$(\bigtriangleup,\bigtriangledown,\bigtriangleup,\bigtriangledown,\bigtriangledown)$	13.00	$(\bigtriangleup,\bigtriangledown,\bigtriangleup,\bigtriangledown,\bigtriangleup)$	13.00
$(\bigtriangleup,\bigtriangledown,\bigtriangleup,\bigtriangleup,\bigtriangledown)$	12.99	$(\bigtriangleup,\bigtriangledown,\bigtriangleup,\bigtriangleup,\bigtriangleup)$	12.00
$(\bigtriangleup,\bigtriangleup,\bigtriangledown,\bigtriangledown,\bigtriangledown)$	12.99	$(\bigtriangleup,\bigtriangleup,\bigtriangledown,\bigtriangledown,\bigtriangleup)$	12.98
$(\bigtriangleup,\bigtriangleup,\bigtriangledown,\bigtriangleup,\bigtriangledown)$	12.91	$(\bigtriangleup,\bigtriangleup,\bigtriangledown,\bigtriangleup,\bigtriangleup)$	12.00
$(\bigtriangleup,\bigtriangleup,\bigtriangleup,\bigtriangledown,\bigtriangledown)$	12.79	$(\bigtriangleup,\bigtriangleup,\bigtriangleup,\bigtriangledown,\bigtriangleup)$	12.00
$(\bigtriangleup,\bigtriangleup,\bigtriangleup,\bigtriangleup,\bigtriangledown)$	12.00	$(\bigtriangleup,\bigtriangleup,\bigtriangleup,\bigtriangleup,\bigtriangleup)$	11.41

La solution avec le polynôme de degré 4 est donc utilisable moyennant un bon choix des coefficients du polynôme d'approximation. Mais on va montrer que même celle de degré 3 l'est en pratique. Ce qui constitue une optimisation significative de l'opérateur mais nécessite des outils.

Le polynôme minimax de degré 3 qui approche le mieux théoriquement 2^x sur $[0,1]$ est:

$$p^*(x) = 0.9998929656 + 0.6964573949x + 0.2243383647x^2 + 0.0792042402x^3.$$

On a alors $\epsilon_{\text{app}} = ||f - p^*||_\infty = 0.0001070344$ soit 13.18 bits de précision. Ceci signifie que, quelle que soit la précision des coefficients utilisés pour représenter p^* et celle utilisée pour son évaluation, on ne pourra pas avoir un opérateur avec une précision meilleure que 13.18 bits.

Afin de déterminer la version représentable de p^* que nous allons implanter, il faut trouver la taille des coefficients. Etant donnés la fonction et son domaine, le format cherché est constitué d'un bit de partie entière et $n-1$ bits de partie fractionnaire. On cherche n minimal pour une erreur d'approximation ϵ_{app} correspondante la plus proche possible du maximum théorique de 13.18.

$n-1$	12	13	14	15	16
ϵ_{app}	12.38	12.45	13.00	13.00	13.02
nb. candidats	0	0	2	2	7

En effet, pour $n-1 = 14$ bits, tous les modes d'arrondis possibles des coefficients donnent:

$(\nabla,\nabla,\nabla,\nabla)$	11.41	$(\nabla,\nabla,\nabla,\triangle)$	12.00
$(\nabla,\nabla,\triangle,\nabla)$	12.00	$(\nabla,\nabla,\triangle,\triangle)$	12.84
$(\nabla,\triangle,\nabla,\nabla)$	12.00	$(\nabla,\triangle,\nabla,\triangle)$	**13.00**
$(\nabla,\triangle,\triangle,\nabla)$	**13.00**	$(\nabla,\triangle,\triangle,\triangle)$	12.36
$(\triangle,\nabla,\nabla,\nabla)$	12.00	$(\triangle,\nabla,\nabla,\triangle)$	12.25
$(\triangle,\nabla,\triangle,\nabla)$	12.23	$(\triangle,\nabla,\triangle,\triangle)$	12.23
$(\triangle,\triangle,\nabla,\nabla)$	12.13	$(\triangle,\triangle,\nabla,\triangle)$	12.12
$(\triangle,\triangle,\triangle,\nabla)$	12.05	$(\triangle,\triangle,\triangle,\triangle)$	11.64

Les deux polynômes candidats sont donc:

$$\frac{8191}{8192}+\frac{2853}{4096}x+\frac{1837}{8192}x^2+\frac{649}{8192}x^3 \qquad \text{et} \qquad \frac{8191}{8192}+\frac{2853}{4096}x+\frac{919}{4096}x^2+\frac{649}{8192}x^3.$$

Tous deux conduisent à une erreur d'approximation de 0.0001220703 (13.00 bits de précision). Il reste maintenant à vérifier que l'évaluation d'au moins un de ces polynômes donne une précision finale d'au moins 12 bits. Voici ci-dessous la précision totale (approximation + évaluation) retournée par GAPPA en utilisant le schéma de Horner et le schéma direct pour évaluer $\frac{8191}{8192}+\frac{2853}{4096}x+\frac{1837}{8192}x^2+\frac{649}{8192}x^3$ pour différentes tailles du chemin de données n':

n'	14	15	16	17	18	19	20
ϵ_{eval} Horner	11.32	11.93	12.36	12.65	12.81	12.90	12.95
ϵ_{eval} direct	11.24	11.86	12.32	12.62	12.79	12.89	12.94

Les valeurs obtenus pour l'autre polynôme ($p_2 = \frac{919}{4096}$) sont équivalentes. Le schéma de Horner présente un comportement légèrement meilleur que l'évaluation directe (qui en plus est plus coûteuse en nombre d'opérations). Il faut un chemin de données sur 16 bits au moins pour obtenir un opérateur avec 12 bits de précision au final et ce à partir d'une approximation avec 13.18 bits de précision.

Passer les coefficients sur 16 bits ne modifie pas beaucoup la précision totale car on voit dans la table qui donne ϵ_{app} en fonction de n que l'on passe de 13.00 à 13.02 bits de précision seulement en passant $n-1$ de 14 à 16 pour l'approximation. Avec des coefficients et un chemin de données sur 16 bits, GAPPA indique une précision de 12.38 bits en évaluant le polynôme $p(x)=\frac{32765}{32768}+\frac{22821}{32768}x+\frac{7351}{32768}x^2+\frac{649}{8192}x^3$.

Deux solutions ont été implantées pour cet opérateur: celle optimisée (degré 3, chemin de données de 16 bits) et celle de base (degré 4, chemin de données de 18 bits). La seconde version correspond à celle que l'on aurait implantée sans l'aide de notre méthode. La table 1 donne les différentes caractéristiques des deux implantations pour un étage de Horner (logique et registres). L'optimisation permet d'obtenir un circuit 17% plus petit mais surtout d'utiliser une approximation de degré 3 plutôt que 4 et donc de gagner 38% en temps de calcul.

version	surface [slices]	période [ns]	nb cycles	durée du calcul [ns]
degré 3, $n' = 16$	193	21.9	3	65.7
degré 4, $n' = 18$	233	26.9	4	107.6

Tableau 1. *Résultats de synthèse pour* 2^x *sur* $[0, 1]$.

4.2. *Racine carrée sur* $[1, 2]$

Pour ce deuxième exemple, nous cherchons à concevoir un opérateur très rapide pour évaluer $\sqrt{x}$ avec x dans $[1, 2]$ et une précision d'au moins 8 bits au final (approximation et évaluation). Le polynôme minimax de degré 1 ne conduit qu'à 6.81 bits de précision, il faut au moins un polynôme de degré 2.

Le polynôme minimax de degré 2 pour $\sqrt{x}$ avec x dans $[1, 2]$ est $0.4456804579 + 0.6262821240x - 0.0711987451x^2$. Il fournit une erreur d'approximation théorique de 0.0007638369 soit 10.35 bits corrects. La précision théorique supérieure à 10 bits permet de supposer que l'on devrait atteindre notre but si on trouve des coefficients représentables sans trop diminuer l'erreur d'approximation.

Toutefois, implanter directement ce polynôme, n'est pas une bonne idée du point de vue du format. Avec x dans $[1, 2]$, il faut travailler un nombre de bits variable pour la partie entière. En effet, l'opération x^2 nécessite deux bits entiers alors que les autres seulement un. Pour éviter ceci, on utilise un changement de variable pour évaluer $\sqrt{1 + x}$ avec x dans $[0, 1]$. Le polynôme minimax correspondant est: $1.0007638368 + 0.4838846338x - 0.0711987451x^2$. On obtient la même erreur d'approximation de 10.35 bits corrects. Ceci est tout à fait normal car le changement de variable $x \leftarrow 1+x$ utilisé ne modifie pas la qualité du polynôme minimax.

A partir de ce polynôme, on pourrait procéder comme pour l'exemple 2^x, mais les coefficients p_0 et p_1 semblent très proches de puissances de 2 et on va essayer de l'utiliser. La première chose à faire est de remplacer p_0 par 1. Le polynôme $1 + 0.4838846338x - 0.0711987451x^2$ offre une précision d'approximation de 9.35 bits, ce qui nous semble satisfaisant.

Le coefficient p_1 semble proche de 0.5. Le polynôme $1 + 0.5x - 0.0711987451x^2$ offre une précision d'approximation de 6.09 bits seulement. p_1 ne peut donc pas être remplacé par 0.5. Toutefois nous allons essayer d'écrire p_1 avec peu de bits à 1 ou -1. Le coefficient p_1 est très proche de $(0.10000\overline{1})_2$. Le polynôme $1 + (0.10000\overline{1})_2x - 0.711987451x^2$ offre une précision d'approximation de 9.45 bits et en plus le produit p_1x est remplacé par la soustraction $\frac{1}{2}x - \frac{1}{2^6}x$.

Nous procédons à une recherche d'une version avec peu de bits non-nuls de p_2 et nous trouvons $(0.0001001)_2$. Donc le produit p_2x^2 est remplacé par l'addition $\frac{1}{2^4}x^2 + \frac{1}{2^7}x^2$. Le polynôme $1 + (0.10000\overline{1})_2x + (0.0001001)_2x^2$ fournit une précision d'approximation de 9.49 bits. Il ne reste donc plus qu'une seule multiplication

pour le calcul de x^2. De plus, on constate qu'avec un coefficient p_2 moins précis (passage de 0.0711987451 à $(0.0001001)_2$), la qualité de l'approximation est légèrement meilleure. Ceci s'explique car le coefficient p_2 quantifié à $(0.0001001)_2$ permet de compenser les erreurs de la quantification des autres coefficients. L'exploration d'une partie de l'espace des coefficients permet de trouver ce genre de polynôme. Mais encore une fois, il s'agit d'une méthode purement heuristique sans aucune garantie ni sur le résultat ni sur le temps de calcul nécessaire pour potentiellement améliorer le résultat. Pour le moment, nous ne voyons pas de méthode pour trouver automatiquement ces petites améliorations.

On va donc déterminer la précision finale intégrant l'erreur d'évaluation en utilisant GAPPA. En cherchant la taille n' du chemin de données on trouve 10 bits. Le programme à faire prouver par GAPPA est le suivant.

```
p0 = 1; p1 = 31/64; p2 = -9/128;
x = fixed<-10,dn>(Mx);
x2  fixed<-10,dn>= x * x;
p   fixed<-10,dn>= p2 * x2 + p1 * x + p0;
Mp = p2 * (Mx*Mx) + p1 * Mx + p0;
{ Mx in [0,1] /\ |Mp-Mf| in [0,0.0013829642]
  -> |p-Mf| in ? }
```

GAPPA retourne une erreur totale de 8.03 bits. Mais ce programme GAPPA correspond à l'utilisation de mutiplieurs pour effectuer les produits $p_1 x$ et $p_2 x^2$. En pratique nous remplaçons ces multiplieurs par des additions/soustractions. Il faut donc donner à GAPPA une description exacte de ce qui est fait par notre architecture. La détermination de la taille minimale du chemin de données est faite en partant de $n = 8$ et en incrémentant n jusqu'à ce que la précision finale soit supérieure ou égale à 8 bits. La recherche donne $n = 13$.

```
p0  = 1;
p11 = 1/2;   p12 = -1/64;
p21 = -1/16; p22 = -1/128;
x = fixed<-8,dn>(Mx);
x2  fixed<-16,dn>= x * x;
p   fixed<-13,dn>= p21 * x2 + p22 * x2 + p11 * x
                   + p12 * x + p0;
Mx2 = Mx * Mx;
Mp  = p21 * Mx2 + p22 * Mx2 + p11 * Mx + p12 * Mx
      + p0;
{ Mx in [0,1] /\ |Mp-Mf| in [0,0.0013829642]
  -> |p-Mf| in ? }
```

Dans ce cas, GAPPA retourne une précision de 8.07 bits avec une seule vraie multiplication pour x^2. L'architecture de l'opérateur est présentée en figure 4. Les cercles gris indiquent un décalage vers la droite du nombre de bits indiqué à l'intérieur du cercle (routage uniquement).

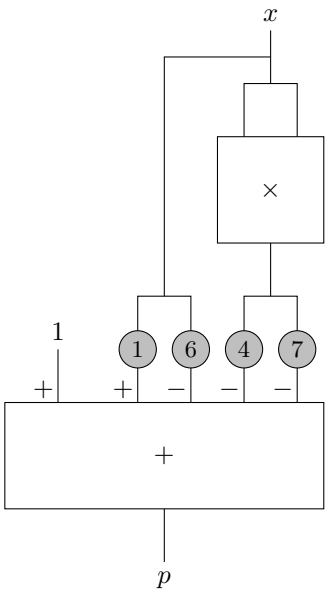


Figure 4. *Architecture de l'opérateur optimisé pour* $\sqrt{1+x}$ *sur* $[0,1]$.

Nous avons implanté la solution optimisée présentée en figure 4 et celle que l'on aurait implantée sans l'aide de la méthode (degré 2, étage de Horner avec chemin de données sur 11 bits). Les résultats de synthèse correspondants sont présentés dans la table 2. Ici aussi, les gains sont intéressants puisque l'on obtient une amélioration de 40% en surface et de 51% en temps de calcul.

version	surface [slices]	période [ns]	nb cycles	durée du calcul [ns]
degré 2 Horner	103	19.9	2	39.8
degré 2 optimisée	61	19.4	1	19.4

Tableau 2. *Résultats de synthèse pour* $\sqrt{1+x}$ *sur* $[0,1]$.

5. Conclusion et perspectives

Nous avons proposé une méthode pour concevoir et optimiser des opérateurs arithmétiques matériels dédiés à l'évaluation de fonctions par approximation polynomiale.

Notre méthode permet, à l'aide d'outils récents, de déterminer une solution avec:

– un degré d petit;

– une taille de coefficients représentables n petite;

– et une taille du chemin de données n' petite.

La méthode ne fournit pas des valeurs de d, n et n' optimales. L'optimum pour ce problème n'est pas connu actuellement même sur le plan théorique. Sur les exemples testés, la méthode permet d'obtenir des circuits plus petits et plus rapides que ceux que l'on pouvait concevoir avant. Toutefois, il reste beaucoup à faire pour les améliorer encore.

De plus, notre méthode permet, grâce à l'utilisation du logiciel GAPPA (Melquiond, 2005–2007), d'obtenir des opérateurs valides numériquement dès la conception. En effet, la méthode permet de déterminer à la fois les erreurs d'approximation et les erreurs d'évaluation. Il n'est plus nécessaire de qualifier *a posteriori* le circuit avec de longues simulations ou tests. La contrainte de précision est vérifiée à la conception.

Dans l'avenir, nous pensons travailler dans plusieurs directions. Pour minimiser l'erreur d'approximation, nous poursuivons nos travaux présentés dans (Brisebarre *et al.*, 2006) et (Brisebarre *et al.*, 2004). Pour minimiser l'erreur d'évaluation tout en évaluant rapidement le polynôme, il reste énormement de travail à faire. En utilisant, par exemple, d'autres schémas d'évaluation de polynômes comme celui proposé par Estrin pour certaines valeurs de d. De plus, toutes les heuristiques présentée pour explorer l'espace des coefficients pour pottentiellement trouver de meilleurs polynômes doivent être encore étudiées. Enfin, nous travaillons sur l'intégration de cette méthode dans des outils pour générer automatiquement des circuits.

Remerciements

Les auteurs tiennent à remercier chaleureusement Guillaume Melquiond pour son aide sur l'utilisation de son outil GAPPA ainsi que les relecteurs pour leur aide précieuse.

6. Bibliographie

Brisebarre N., Muller J.-M., Tisserand A., « Sparse-Coefficient Polynomial Approximations for Hardware Implementations », *Proc. 38th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, U.S.A., p. 532-535, November, 2004.

Brisebarre N., Muller J.-M., Tisserand A., « Computing Machine-Efficient Polynomial Approximations », *ACM Transactions on Mathematical Software*, vol. 32, n° 2, p. 236-256, June, 2006.

Chesneaux J.-M., Didier L.-S., Jézéquel F., Lamotte J.-L., Rico F., « CADNA: Control of Accuracy and Debugging for Numerical Applications », , http://www-anp.lip6.fr/cadna/, n.d. LIP6–Univ. Pierre et Marie Curie.

Cordesses L., « Direct Digital Synthesis: A Tool for Periodic Wave Generation (Part 1) », *IEEE Signal Processing Magazine*, vol. 21, n˚ 4, p. 50-54, July, 2004.

Ercegovac M. D., Lang T., *Digital Arithmetic*, Morgan Kaufmann, 2003.

Goubault E., Martel M., Putot S., « FLUCTUAT: Static Analysis for Numerical Precision », , http://www-list.cea.fr/labos/fr/LSL/fluctuat/, n.d. CEA-LIST.

Melquiond G., « GAPPA: génération automatique de preuves de propriétés arithmétiques », , http://lipforge.ens-lyon.fr/www/gappa/, 2005–2007. Arénaire, LIP, CNRS-ENSL-INRIA-UCBL.

Ménard D., Sentieys O., « Automatic Evaluation of the Accuracy of Fixed-Point Algorithms », *in* C. D. Kloos, J. da Franca (eds), *Proc. Design, Automation and Test in Europe (DATE)*, p. 529-537, March, 2002.

Muller J.-M., *Elementary Functions: Algorithms and Implementation*, 2nd edn, Birkhäuser, 2006.

Remes E., « Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation », *C.R. Acad. Sci. Paris*, vol. 198, p. 2063-2065, 1934.